UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Migration of networks in a multi-cloud environment

José Pedro Coelho Soares

MESTRADO EM ENGENHARIA INFORMÁTICA
Arquitetura, Sistemas e Redes de Computadores

Dissertação orientada por:
Prof. Doutor Fernando Manuel Valente Ramos

2018

Agradecimentos

Quero agradecer aos meus pais por toda a motivação que sempre me deram, e pela força que me transmitiram para ultrapassar os vários desafios que me foram surgindo. Sem o seu apoio esta viagem teria sido mais difícil. Agradeço também aos meus avós pelo apoio e carinho que sempre me transmitiram.

Não posso também deixar de agradecer aos meus colegas Joel Alcantara, Frederico Brito e Luis Ferrolho pela disponibilidade que sempre demonstraram e pelos momentos que partilhamos durante o curso. Partilhamos momentos de muito trabalho mas também momentos de boa disposição.

Por fim, ao meu orientador, Prof. Doutor Fernando Manuel Valente Ramos, pelos conhecimentos transmitidos, pelo papel fundamental que representou na minha formação profissional e pela sua disponibilidade.

A todos muito Obrigado.

À minha família e amigos.

Resumo

A forma como os centros de dados e os recursos computacionais são geridos tem vindo a mudar. O uso exclusivo de servidores físicos e os complexos processos para provisionamento de software são já passado, sendo agora possível e simples usar recursos de uma terceira parte a pedido, na nuvem (cloud).

A técnica central que permitiu esta evolução foi a virtualização, uma abstração dos recursos computacionais que torna o software mais independente do hardware em que é executado. Os avanços tecnológicos nesta área permitiram a migração de máquinas virtuais, agilizando ainda mais os processos de gestão e manutenção de recursos.

A possibilidade de migrar máquinas virtuais libertou o software da infraestrutura física, facilitando uma série de tarefas como manutenção, balanceamento de carga, tratamento de faltas, entre outras. Hoje em dia a migração de máquinas virtuais é uma ferramenta essencial para gerir clouds, tanto públicas como privadas.

Os sistemas informáticos de grande escala existentes na cloud são complexos, compostos por múltiplas partes que trabalham em conjunto para atingir os seus objectivos. O facto de os sistemas estarem intimamente ligados coloca pressão nos sistemas de comunicação e nas redes que os suportam. Esta dependência do sistema na infraestrutura de comunicação vem limitar a flexibilidade da migração de máquinas virtuais. Isto porque actualmente a gestão de uma rede é pouco flexível, limitando por exemplo a migração de VMs a uma subrede ou obrigando a um processo de reconfiguração de rede para a migração, um processo difícil, tipicamente manual e sujeito a falhas.

Idealmente, a infraestrutura de que as máquinas virtuais necessitam para comunicar seria também virtual, permitindo migrar tanto as máquinas virtuais como a rede virtual. Abstrair os recursos de comunicação permitiria que todo o sistema tivesse a flexibilidade de ser transferido para outro local.

Neste sentido foi recentemente proposta a migração de redes usando redes definidas por software (SDN), um novo paradigma que separa a infraestrutura de encaminhamento (plano de dados) do plano de controlo. Numa SDN a responsabilidade de tomar as decisões de controlo fica delegada num elemento logicamente centralizado, o controlador, que tem uma visão global da rede e do seu estado. Esta separação do plano de controlo do processo de encaminhamento veio facilitar a virtualização de redes.

No entanto, as recentes propostas de virtualização de redes usando SDN apresentam

limitações. Nomeadamente, estas soluções estão limitadas a um único centro de dados ou provedor de serviços. Esta dependência é um problema. Em primeiro lugar, confiar num único provedor ou cloud limita a disponibilidade, tornando efectivamente o provedor num ponto de falha único. Em segundo lugar, certos serviços ficam severamente limitados por recorrerem apenas a uma cloud, devido a requisitos especiais (de privacidade, por exemplo) ou mesmo legais (que podem obrigar a que, por exemplo, dados de utilizadores fiquem guardados no próprio país). Idealmente, seria possível ter a possibilidade de tirar partido de *múltiplas clouds* e poder, de forma transparente, aproveitar as vantagens de cada uma delas (por exemplo, umas por apresentarem custos mais reduzidos, outras pela sua localização). Tal possibilidade garantiria uma maior disponibilidade, visto que a falha de uma cloud não comprometeria todo o sistema. Além disso, poderia permitir baixar os custos porque seria possível aproveitar a variação dos preços existente entre clouds ao longo do tempo. Neste contexto multi-cloud um dos grandes desafios é conseguir migrar recursos entre clouds de forma a aproveitar os recursos existentes. Num ambiente SDN, em particular, a migração de redes é problemática porque é necessario que o controlador comunique com os elementos físicos da rede para implementar novas políticas e para que estes possam informar o controlador de novos eventos. Se a capacidade de comunicação entre o controlador e os elementos de rede for afectada (por exemplo, devido a latências elevadas de comunicação) o funcionamento da rede é também afectado.

O trabalho que aqui propomos tem como objectivo desenvolver algoritmos de orquestração para migração de redes virtuais, com o objectivo de minimizar as latências na comunicação controlador-switches, em ambientes multi-cloud. Para esse efeito foi desenvolvida uma solução óptima, usando programação linear, e várias heurísticas. A solução de programação linear, sendo óptima, resulta na menor disrupção possível da ligação ao controlador. No entanto, a complexidade computacional desta solução limita a sua usabilidade, levando a tempos de execução elevados. Por esta razão são prospostas heurísticas que visam resolver o problema em tempo útil e de forma satisfatória. Os resultados das nossas experiências mostram que nas várias topologias testadas algumas heurísticas conseguem resultados próximos da solução óptima. O objectivo é atingido com tempos de execução consideravelmente inferiores.

Palavras-chave: virtualização, cloud, redes de computadores, migração

Abstract

The way datacenters and computer resources are managed has been changing, from bare metal servers and complex deployment processes to on-demand cloud resources and applications.

The main technology behind this evolution was virtualization. By abstracting the hardware, virtualization decoupled software from the hardware it runs on. Virtual machine (VM) migration further increased the flexibility of management and maintenance procedures. Tasks like maintenance, load balancing and fault handling were made easier.

Today, the migration of virtual machines is a fundamental tool in public and private clouds. However as VMs rarely act alone, when the VMs migrate, the virtual networks should migrate too. Solutions to this problem using traditional networks have several limitations: they are integrated with the devices and are hard to manage. For these reasons the logical centralisation offered by Software-Defined Networking (SDN) architectures has been shown recently as an enabler for transparent migration of networks.

In an SDN a controller remotely controls the network switches by installing flow rules that implement the policies defined by the network operator. Recent proposals are a good step forward but have problems. Namely, they are limited to a single data center or provider. The user's dependency on a single cloud provider is a fundamental limitation. A large number of incidents involving accidental and malicious faults in cloud infrastructures show that relying on a single provider can lead to the creation of internet-scale single points of failures for cloud-based services.

Furthermore, giving clients the power to choose how to use their cloud resources and the flexibility to easily change cloud providers is of great value, enabling clients to lower costs, tolerate cloud-wide outages and enhance security. The objective of this dissertation is therefore to design, implement and evaluate solutions for network migration in an environment of multiple clouds. The main goal is to schedule the migration of a network in such a way that the migration process has the least possible impact on the SDN controller's ability to manage the network. This is achieved by creating a migration plan that aims to minimize the experienced control plane latency (i.e., the latency between the controller and the switches). We have developed an optimal solution based on a linear program, and several heuristics. Our results show that it is possible to achieve results close to the optimal solution, within reasonable time frames.

Keywords: virtualization, clouds, computer networks, migration

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The management of computational resources has been changing fast. First, we had physical servers where applications were deployed. Then, virtualization was introduced [7], with software solutions no longer tied to specific hardware. Virtualization allowed to interchangeably run any application on any hardware, resulting in benefits like the possibility to balance load, perform maintenance, prepare for disasters and increase fault-tolerance.

With virtualization already in place, virtual machine migration [12] was proposed and further enhanced the way datacenters are managed, from the ability to relocate resources to enabling zero downtime hardware maintenance. Virtual machine (VM) migration is a fundamental tool in cloud operations nowadays. With the emergence of multiple cloud infrastructures another opportunity arises: the possibility to leverage resources from multiple cloud operators to lower costs and improve reliability.

1.1 Motivation

Today's computer applications consist of distributed components that work together to provide a variety of services. Without stable communication the applications and the systems that support them are rendered useless. Therefore, it's important to ensure that both the virtual machines (VMs) and their ability to communicate are functioning properly. One problem that limits VM migration is therefore its tight integration with the underlying network resources. For instance, a VM cannot move to arbitrary locations due to the coupling of network addressing with a VM's physical location. As such, virtual machine migration is usually limited to the same subnetwork or might require planning due to the fact that the network needs reconfiguration, since network policies (quality of service and isolation for example) may need the be reconfigured since they are specific to a physical network.

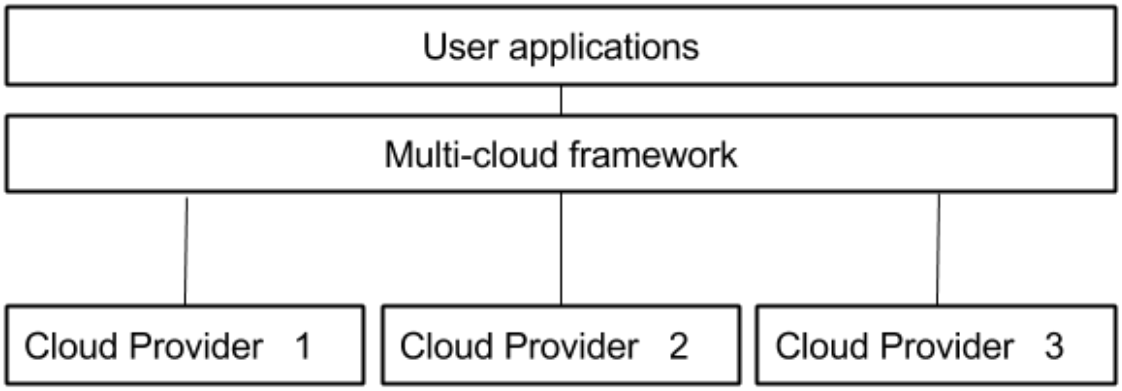
Ideally, when moving the state of a particular application VM, the network interconnecting them should move with it. However, this tight coupling interferes with VM migration and calls for network virtualization techniques to provide the needed flexibility

[23].

Much like the VM is itself a virtualization of computational resources, networks should be virtualized, isolated from each other and detached from their physical infrastructure to facilitate network migration. By *network migration* we mean the ability to migrate network state – that can include forwarding rules, metadata associated to packets, etc. – from one switch instance to another. The separation of the control and data planes and the logical centralization of control in a cluster of servers achieved by Software-Defined Networking (SDN [25]) now give operators the ability to virtualize networks at cloud scales. Recently proposed network virtualization techniques [23] use SDN to offer full network virtualization.

Existing solution are usually restricted to a single cloud provider, limiting dependability and scalability. By contrast, a flexible solution (Figure 1.1) that could leverage cloud resources would grant users several benefits. First, compliance with privacy legis-

Figure 1.1: A multi-cloud solution



lation may demand certain customer data to remain inside a country. As such the ability to have a network span multiple clouds (keeping user data locally) would allow to fulfill these obligations but still leverage cloud resources to scale out. Second, as availability is an issue (a single cloud is a single point of failure), a multi-cloud solution would be more resilient to outages, enabling a service to withstand a datacenter and cloud outages. Unfortunately this problem is increasingly more common [3]. Third, user costs can potentially be decreased by taking advantage of pricing plans from multiple cloud providers: a multi-cloud platform could migrate resources between clouds, to reduce operational costs. Fourth, performance can also be enhanced by bringing services closer to clients or by migrating VMs that at a certain point in time need to closely cooperate. As such, creating solutions that provide more flexibility regarding the way users can leverage resources from multiple clouds is considered increasingly relevant [5]. The main motivation for this work is thus to explore on how to extend SDN-based network migration to this

multi-cloud environment.

1.2 Goals and challenges

In this work we aim to create solutions for migration of networks in a multi-cloud SDN-based environment. We target a multi-cloud network virtualization platform such as [6]. This entails two fundamental challenges:

- Working outside the boundaries of a datacenter requires the use of low bandwidth and high latency links, representing a complicated environment when compared to the use of fast internal datacenter connections of single cloud solutions.

- In an SDN scenario the control depends on the network controller and its interactions with switches. If the communication with the controller is affected, so is the network's ability to react to network events and implement new policies. Therefore, it's important to ensure that the communication between switches and the controller (i.e the control plane) are kept as stable as possible.

A multi-cloud network migration solution will have to face these issues to be useful and practical. The goal of this work is thus to investigate network migration orchestration algorithms with the goal of minimizing downtime of the control plane communications and thus assuring a smooth migration.

1.3 Contributions

In order to ensure smooth network migration in a multi-cloud SDN the connection between the switches and the network controller must be kept stable, particularly in terms of reduced latency. The particular order in which network elements are migrated affects control plane latencies, and as such it is important to plan the migration order (i.e., to orchestrate the migration) so that any potential disruption is minimized. To address this problem this work makes the following contributions:

- We propose an optimal solution for the network migration problem using a linear programming formulation.

- As the optimal solution is computationally intractable, we propose heuristics to solve the problem.

- We have performed a thorough evaluation of the proposed solutions considering a diverse set of network topologies. The conclusion is that some heuristics achieve solutions close to the optimal within reasonable time bounds.

1.4 Structure of the document

This document is organised as follows: In section 2 we present the state of the art, including an introduction to network virtualization and live migration of virtual networks. In section 3 we present the proposed solutions for orchestration of SDN migration. In Section 4 we evaluate the proposed algorithms, and in section 5 we conclude this dissertation.

Chapter 2

Related Work

The need for more flexibility in the way computational resources are managed is not new. Several technologies (virtualization, migration, etc.) have been developed in the past decade for that purpose, and they have changed the way resources are used. In this chapter, we discuss techniques that have made the process of managing resources easier, increasing flexibility. We start by discussing virtualization: the challenges it addresses, and several types of virtualization mechanisms and their benefits. Then, live migration of virtual machines is introduced: we address the need for this mechanism and its main limitations.

We then change the focus to the networking aspect we target. We present software-defined networking (SDN), discussing the architecture, related standards and address some challenges such as scalability. Next, we introduce network virtualization and discuss its benefits. Then, we build upon the previous section to discuss network migration.

2.1 Virtualization

Virtualization technology consists in abstracting the underlying hardware to the applications. In practice this means that the software is no longer tied to the hardware, making the software more independent from the infrastructure it is executed on. The ability to run virtualized operating systems and workloads provided datacenter operators with more flexibility and efficiency in the way they manage their computational resources. Advantages of virtualization include:

1. Easier deployment of applications, made possible by bundling the application inside a virtual machine and running it on any existing physical server without having to reconfigure it.

2. Deployment failures are also reduced as it is possible to make sure that the testing environment and the production environment match, meaning less unexpected errors when the time to deploy the software arrives.

3. Reduced hardware vendor lock-in: since the software is designed to run on a VM any hardware that can run the Virtual Machine Manager (VMM) is suitable.

4. Resource isolation, since VMs are executed on their own instances of the operating system and the VMM guarantees that each VM gets the resources it needs without affecting the others.

Over the years different virtualization techniques have been developed. Some offer full virtualization, requiring no changes in the guest OS, while others require changes in applications or/and operating system.

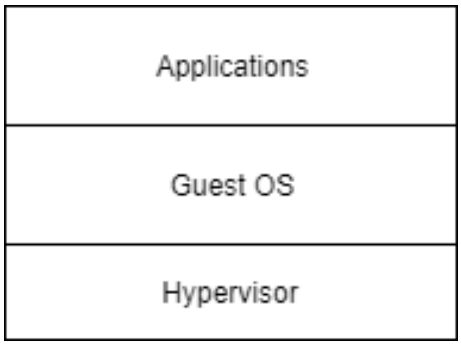


Figure 2.1: Hypervisor based virtualization

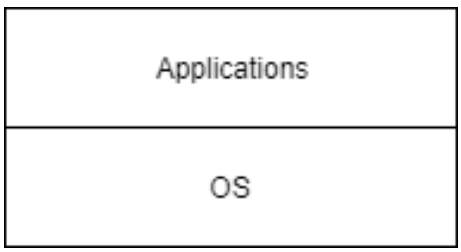


Figure 2.2: Kernel-based virtualization

Another key difference is how virtualization is achieved [1]: some solutions (Figure 2.1) introduce a new software layer, called hypervisor, while others (Figure 2.2) work at the operating system level to include mechanisms that allow for virtualization (such as ensuring isolation). The first, full virtualization (as offered by Xen [7]), is more expensive than the second, kernel-based virtualization (such as linux containers [1]), since in the former a guest VM runs an entire operating system on top of the virtualized resources, whereas on the latter there is only one operating system that hosts all the virtual machines.

Choosing between different types of virtualization requires knowing the problem well.

For example, a cloud provider may opt for a virtualization solution with strong guarantees of isolation, so that clients do not interfere with each other's resources. On the other hand, a virtualization solution for a private cloud could be more lightweight as it would still obtain the flexibility of a virtualized environment without the overhead of full virtualization.

Under a multi-cloud context as the one we target, another type of virtualization is helpful: nested virtualization (Figure 2.3). Nested virtualization consists of running a second hypervisor on top of the cloud provider's hypervisor. The end result is hypervisor level control on third party clouds, an important tool to create multi-cloud solutions. Nested virtualization enables the implementation of new features without provider support. For instance, VM migration might not exist in a particular cloud solution, but by leveraging nested virtualization such feature can be implemented by the user's hypervisor. This flexibility is valuable to solve cloud heterogeneity issues and to enable new services on top of existing cloud offerings, such as cloud fault tolerance.

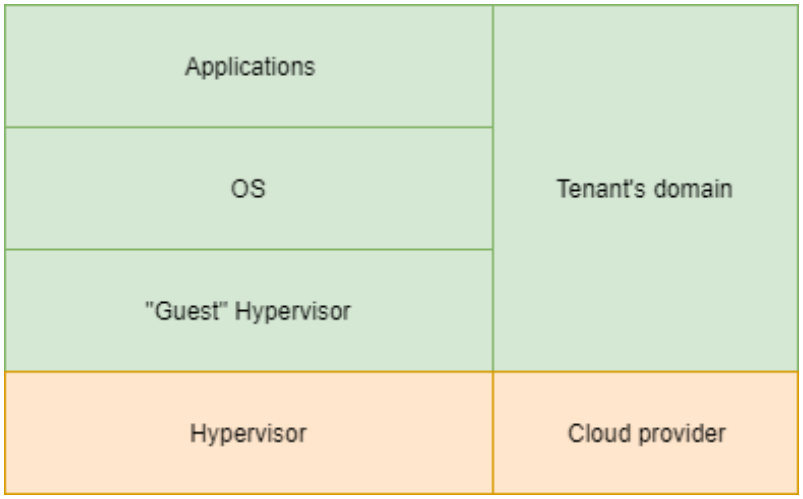


Figure 2.3: Nested virtualization

A related problem is that today's clouds operate in isolation, have differences regarding their VM format and the features they provide. All this heterogeneity makes it very hard to create solutions that can span multiple clouds. As an example of a solution that offers a service that spans multiple clouds is Xen-Blanket [34] (Figure 2.4). This system provides users total control of their resources that span multiple clouds. This is achieved by enabling a hypervisor-level of control on third-party clouds and providing a consistent set of features in every cloud. It includes a layer responsible for hiding the heterogeneity of clouds (Blanket layer), since different clouds provide different interfaces for accessing resources (such as networking and disk I/O). For this purpose, this layer includes drivers

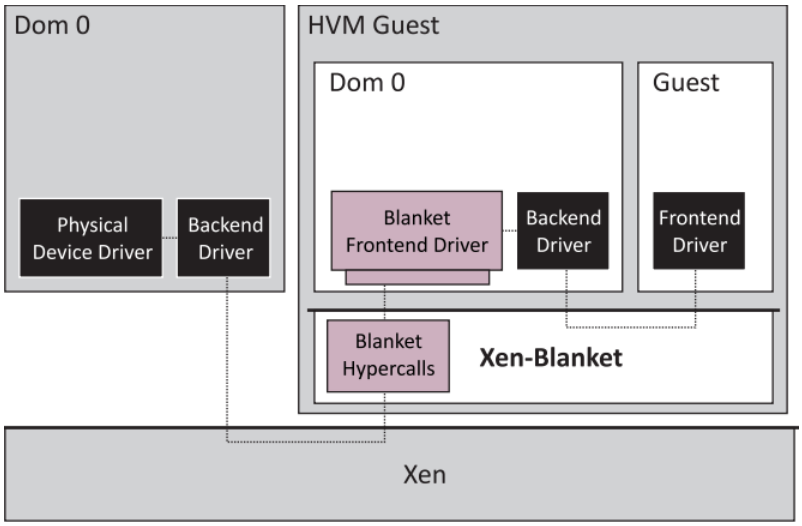


Figure 2.4: Xen-Blanket architecture, from [34]

for every cloud interface, called Blanket Drivers. The use of these drivers makes it possible to present a consistent interface to guests, independently of the cloud they are in.

2.2 Live Migration

Managing data centers and computer clusters is hard: there is the need to continuously support new applications, hardware has to be replaced or upgraded periodically, and the available resources have to be used efficiently. Traditional solutions to these problems included planned maintenance schedules to replace hardware and administrative tasks such as re-arranging resource allocations. These solutions usually incurred in downtimes which could range from minutes to hours.

To address these issues it is now possible to use live virtual machine migration. With live migration administrators are given more flexibility to manage their resources. Tasks like maintenance, load balancing and fault-management no longer need to cause long periods of downtime, as it is possible to migrate VMs to another server live, by re-assigning VMs to physical machines [12].

2.2.1 Live Migration of Virtual Machines

The work by Clark et al. [12] addresses the problem of implementing a live migration mechanism. The proposed solution targets a local-area network, and it builds upon the Xen hypervisor. By leveraging Xen's virtualization the authors implemented a solution that does not require any participation of the guest OS. The proposed method works in multiple stages. First, a target machine is selected to host the VM, and as such it is necessary to ensure that sufficient resources are available. Then resources are reserved at the target machine. The second stage consists of an iterative pre-copy phase, where all the memory pages are sent to the target machine. In every iteration only the pages that

have been changed since the last iteration need to be sent again. This iterative behaviour continues until a point where the same group of memory pages are the only ones being modified (these pages are called the writable working set). When such situation arises, we enter the stop-and-copy phase where the VM at the original host is suspended and these pages are transferred to the destination machine. Finally, the target machine acknowledges that it received a consistent OS image, sends an ARP reply so that packets are sent to the new location and resumes the VMs execution.

The process of determining the writable working set is rather important because a correct decision on when to stop the iterative pre-copy phase will optimize both network usage and total migration time. To develop heuristics to determine when to start the final stop-and-copy phase the authors analyzed several workloads and bounded the number of pre-copy rounds to these results. To facilitate an efficient use of network resources the migration routine proposed also adapts its use of bandwidth. When all that is left for the migration to finish is the working set the bandwidth use is increased to finish.

2.2.2 Live WAN migration of VMs

Just like local-area migration changed the way resources are handled in a datacenter, WAN migration can possibly provide the same flexibility but on a global scale.

Several issues arise when trying to perform live migration on a wide-area network. First, there are bandwidth restrictions. Second, the latency is higher. Finally, the virtual machine changes to a different layer 3 network, which means the VM has to change its IP address. To address these challenges, in 2010 Timothy Wood and K.K. Ramakrishnam presented CloudNet [35], a platform for optimized WAN migration of Virtual Machines. The authors present the notion of a Virtual Private Cloud, which represents a set of resources securely and transparently connected to the existing infrastructure. To provide transparent networking CloudNet uses a Multiprotocol Label Switching (MPLS) based virtual private network (VPN) and Virtual Private LAN Services (VPLS [21]), creating the abstraction that all the resources exist in the same network while in reality they reside in different networks. With this type of network virtualization in place the IP address problem is solved. In addition, VPLS creates a shared Ethernet broadcast domain, and by doing so the unsolicited ARP message that is sent to update the network about the VM's new location [12] reaches all the sites connected through the VPN, updating the whole network about the VM's new location.

To perform an efficient WAN migration, CloudNet proposes some changes to the standard migration routine to better fit a WAN scenario. The migration algorithm in Xen has a set of conditions to determine when to proceed to the final migration steps. Specifically, Xen enters this final step if one of three conditions is met: a) either there is a very small number of pages remaining, or b) three times the VM's memory has been sent, or c) more than 30 iterations took place.

The authors of [35] found that even under an average workload the condition that was met more commonly was the latter. Their results also shown that most of these iterations where wasteful and only increased the migration time and bandwidth used. To address this, they proposed a different heuristic to determine when to end the iterative page copying stage. Their proposal tracks the number of pages remaining to be sent in a short history. If there are fewer pages to send than any entry recorded in the history, they enter the final iteration (in order to prevent their optimization from performing poorly, if there is an increasing trend in the number of remaining pages the migration is terminated early, should such trend be detected). In addition, aiming to adapt to the lower bandwidth scenario, CloudNet uses content based redundancy using a block based scheme. Simply put, this divides the content (RAM and disk) in fixed size blocks and calculates a hash for every block. When sending data over the network if both the source and destination have the hash for this block in their caches it means they can send a small 32 bit index for the cache entry instead of sending the actual block, saving bandwidth.

A different approach to enable WAN migration is proposed in [10]. The goal is to migrate a VM and its local storage across a WAN without losing ongoing connections. To accomplish this goal the authors employed different solutions: dynamic DNS and tunnelling. First, to ensure that the connections stay alive a tunnel is created between the original physical VM host and the target VM host to relay the connections and ensure that they are not lost. To ensure that new connections are sent to the new IP address the authors leveraged dynamic DNS. The idea is to update a VM's DNS entry right before it migrates so that new connections are sent to its new location.

To ensure a faster WAN migration the authors proposed a series of techniques. First, the use of image templates to reduce the amount of data that has to be sent across the WAN links. When a VM migrates, the only data that needs to be transferred is the difference between the template and the actual VM image, saving bandwidth. Second, considering that some workloads might be write-intensive (which could delay the migration process), a write-throttling mechanism was implemented where after a certain pre-defined threshold is exceeded, all successive write operations are delayed. Without this mechanism some workloads could prove to be very difficult to migrate in a timely fashion. The experimental results show that by using this technique the service disruptions caused by migration are unnoticeable by end-users.

2.2.3 Migration for fault-tolerance

Datacenters are designed to account for faults by employing redundant architectures that account for servers becoming unavailable. These replication solutions aim to increase a service's availability, as power equipment failures and other problems often make tens to thousands of servers unavailable and downtime is expensive. [3]

Unfortunately, there is a tradeoff between fault tolerance and reducing bandwidth us-

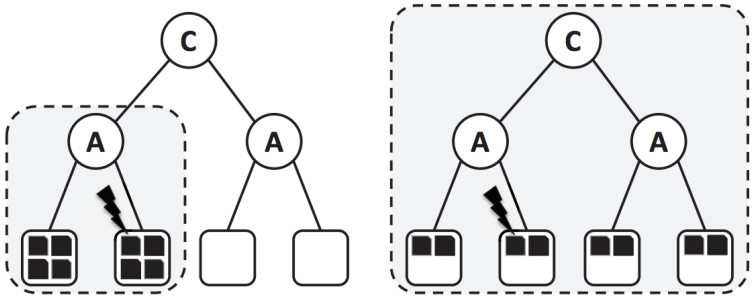


Figure 2.5: Bandwidth usage and Fault tolerance, from [9]

age [9]: to lower bandwidth one can deploy the servers together but this reduces fault-tolerance, because one fault can bring an entire service down. On the other hand, spreading the servers increases their fault-tolerance but increases bandwidth usage (Figure 2.5). The problem of optimizing fault tolerance and bandwidth usage is both NP-Hard and hard to approximate. Considering this, the authors of [9] formulated a convex optimization problem that incentivizes spreading machines but penalizes machine allocations that increase bandwidth usage. For this purpose the authors examined the traffic patterns of a real large-scale web application. This study provided important insights about communication patterns and the impact of faults. The authors discovered that:

- The network core is highly used;
- The traffic matrix is very sparse;
- Only 2% service pairs communicate at all, and the communication pattern of these 2% is very skewed;
- 1% of the services generate 64% of all the traffic;
- The median service talks to 9 other services;
- Networking and hardware failures can cause significant outages;
- Power faults domains create non-trivial patterns.

The authors leveraged these insights to design algorithms that can account for both fault tolerance and bandwidth. The optimization framework takes into account the following metrics: bandwidth, fault tolerance, and the number of moves (the number of servers that would have to be relocated to move from the starting configuration to the optimized one). They proposed the following algorithms: a) CUT+FT, where there is no penalty for using bandwidth, b) CUT+FT+BW, where machine placements that increase bandwidth

usage suffer penalties. Both algorithms work in two phases. The first phase places machines minimizing bandwidth usage. Then, in the second phase, machines are swapped around to improve fault-tolerance. The authors evaluated their algorithms and determined CUT+FT+BW performs the best. Their solution achieved 20%-50% reduction in bandwidth usage in the core of the network and still improved the average worst-case survival by 40%-120%.

Traditional methods for making fault-tolerant applications involve undertaking the complex task of creating complex recovery routines on otherwise simple applications, making the process of creating and deploying such applications harder. To address this problem Remus [13] is a mechanism that provides fault-tolerance as a service, enabling the creation of highly available systems without modifications to the original application and without complex fault-tolerant software. Remus works by replicating several times per second a VM's internal state to another physical machine (thus migrating its state to the replica). When the original machine fails, the backup can take its place. To avoid severe performance degradation from executing both hosts in lock-step, Remus allows the first host to execute speculatively and replicate its state asynchronously to the backup machine. This, however, would cause consistency issues should the primary reply to a client with a confirmation and then crashed, as the backup machine might not had received a snapshot with that operation. To ensure consistency some care was necessary. Namely, Remus does not allow the outside world to view state unless such state has already been successfully replicated. In practice, Remus buffers all network output until a confirmation that the checkpoint has been replicated arrives. When that notification arrives the state associated with the checkpoint is released, meaning that when a client receives a reply the associated state has already been replicated, and by doing so ensuring consistency.

2.2.4 Algorithms for placement of virtual machines

Virtualization and live migration can incur in inefficient usage of resources if used carelessly. Unplanned VM assignment or bad placement could lead to overloaded machines and network congestion. Properly consolidating machines is therefore beneficial.

Towards this end the authors of [4] present a heuristic for consolidating heterogeneous VMs by considering their communication graph. They design an algorithm to place VMs in a way that minimizes traffic between VMs hosted in different physical machines. The proposed heuristics make the following assumptions: first, that inter-tenant VM communication is either small or non- existent. Second, that during off-peak hours it is valid to consolidate all of a tenant's VMs in a single machine. The proposed algorithm works in the following way: First, it determines which machines are loaded beyond a given threshold. These are considered unsuitable targets for migration. Then, it determines which machines are loaded below a given second threshold. These are the actual migration targets. This second set of machines is sorted by load. Afterwards, a set of VMs whose

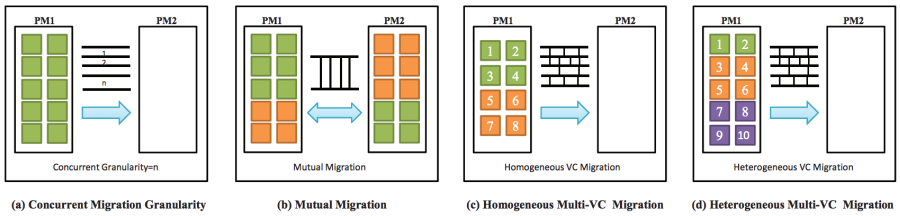


Figure 2.6: Migration strategies, from [4]

load can be accommodated by these physical machines is identified. Finally, using the communication graph (of the VMs they want to migrate) they try to migrate the largest connected groups of VMs to a single machine. At some point it will be impossible to place an entire group of VMs in a single physical machine. When that situation arises this group of machines is partitioned into smaller groups. For this purpose the authors present two partitioning algorithms, a modified breadth first search and a modified Prim's algorithm.

One problem of traditional migration techniques is that they are insufficient when an entire virtual cluster or multiple virtual clusters need to be migrated. Considering this, the authors of [36] analyzed various live migration strategies for virtual clusters (VC). The proposed VC-Migration framework is responsible for managing the VC migrations. This solution monitors the resource usage (CPU, memory, disk I/O, and network) the address the migration's performance. The authors have considered 4 migration strategies: a) Concurrent migration, with a varying number of machines being migrated at the same time; b) mutual migration, which consists of two clusters on two physical machines migrating to each other's physical machine simultaneously; c) homogeneous multi-VC migration and d) heterogeneous multi-VC migration, which consists of migrating multiple clusters of the same size, and multiple clusters with different sizes, respectively. This is shown in Figure 2.6.

This work analyzed these migration strategies and determined their effectiveness regarding multiple parameters, including migration time and bandwidth usage. Their conclusions where the following:

- Virtual machines with more memory increase migration time. More memory results in a longer pre-copy phase, therefore increasing the total migration time. However the downtime is not affected, since it depends on the page dirting rate and transfer speed [10].

- Concurrent migration of large numbers of machines severely hurts performance. The limited network bandwidth slows the migration process for large numbers of concurrent migrations. For the same reasons (limited bandwidth) sequential migration is better than mutual migration.

- Virtual machines belonging to the same virtual cluster should be deployed together,

to reduce the communication and synchronization latency across different physical machines.

- Mutual migration should be avoided due to the long overall migration time.
- Migration order is important when multiple virtual clusters need to be migrated.

These results are very important. Although they detail strategies focused on a different problem (migration of VM, not networks), they provide useful insights to our work, such as the importance of migration order. This demonstrates the need for effective orchestration algorithms, and is thus a significant motivation for our work.

After discussing virtualization and migration of virtual machines, in the next section we focus on networks.

2.3 Software-defined networking

The Internet has changed the way computer systems (and people!) communicate. It has been doing so for years. Despite their undeniable success, traditional IP networks are complex and hard to manage [25]. Indeed, correctly applying a policy into hundreds or thousands of network devices using manual or low-level scripting methods is not an easy task. Neither is constantly changing the network to adapt to new workloads. To further complicate the matter, current networks are vertically integrated. Its several layers (data, control, management) are tightly coupled inside vendor-specific, closed software and hardware.

New standards take years to be agreed upon, and when finalized the implementations are typically closed, which means they cannot be reused nor modified. As a result, computer networks have evolved slowly.

For example, the transition from IPv4 to IPv6, despite its relevance and the fact that it started over 20 years ago, has yet to reach completion.

2.3.1 Architecture

Software-Defined Networking (SDN) is an architecture that emerged with the aim to fix the shortcoming of traditional networks. In an SDN the data plane and the control plane are decoupled, meaning that they are not integrated in the same device. The data plane functions run on switches, while the control plane logic is executed in an external entity called controller. This separation of concerns is one of the defining characteristics of an SDN. The network equipment (switches) takes care of forwarding the traffic and the controller is responsible for the control plane, that is, for deciding how to forward traffic. The control plane is executed in commodity hardware, typically in a cluster of regular servers. This controller is usually thought of as “logically centralized” as it offers

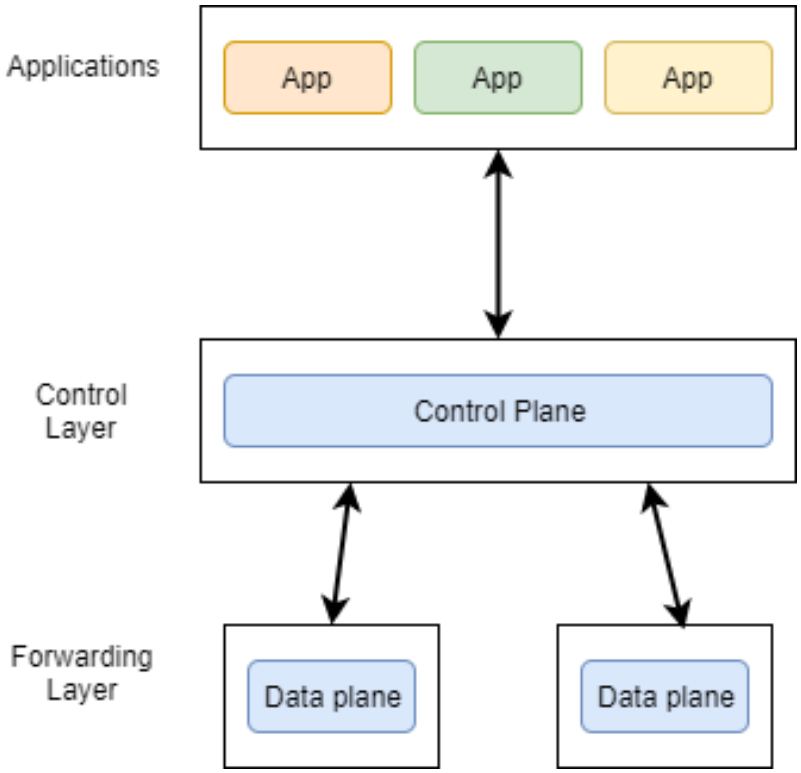


Figure 2.7: SDN Architecture

a global view of the system and the ability to control it in a much more consolidated way, but in practice its implementation can be physically distributed.

Figure 2.7 illustrates the SDN architecture. In the bottom we have the data plane. The connection between the switches (data plane) and the controller (control plane) is referred as the southbound API (the most common being the Openflow [28] protocol). The connection between the controller and the applications is called the northbound API. In short, network applications have a high level abstract view of the network and implement their policies there. These policies are then translated into a set of switch rules by the control plane, and finally these rules are applied on the actual hardware through the southbound API. Assuming that the APIs remain the same all these parts can move independently: it's possible to deploy faster switches with little to no reconfiguration or without reprogramming an application. To address scalability issues it is possible to distribute the controller and use sharding techniques without changing the other components.

2.3.2 OpenFlow

As stated above the most common southbound API for an SDN is OpenFlow [28, 25]. The goal of a southbound API is to hide the heterogeneity between network devices, and normalize the way their behaviour is controlled by means of a common interface. In OpenFlow the way to control forwarding is by defining rules to match packets, and actions to execute on packets which headers matched (Figure 2.8).

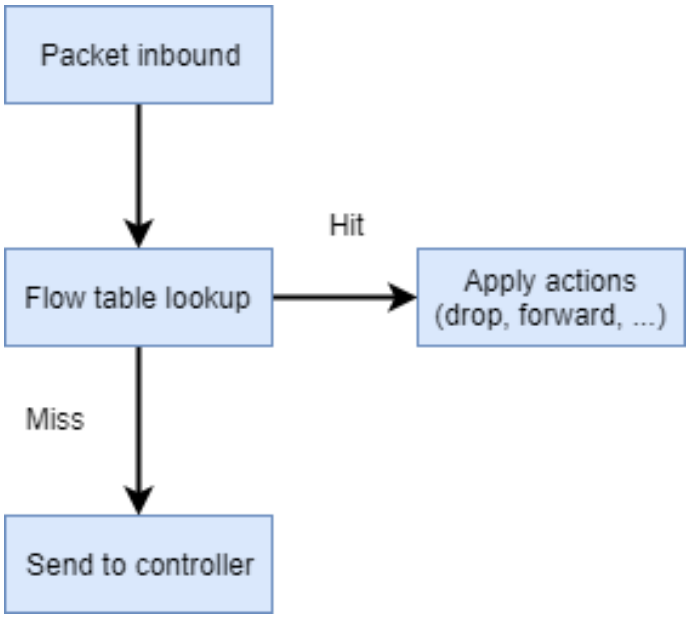


Figure 2.8: Simplified view of Openflow handling a packet

When a packet is received, the switch looks-up a forwarding table that contains several rules to be verified. If that packet header matches some rule, the corresponding actions are applied to the packets for that flow. Actions include dropping the packet, forwarding it to some port(s), sending it to the controller or to another table, among others. To perform the matching several header fields can be used. These include TCP, IP, VLAN, Ethernet, etc. Rules also have an associated priority, dictating the order by which they should be evaluated. Several possible execution modes exist. In reactive mode, the first packet from a given flow is sent to the controller as a packet-in event. The controller then installs the in the switch the necessary rules so that subsequent packets from that flow do not leave the data plane. In proactive mode the controller populates the switches with forwarding rules ahead of time, not waiting to be triggered by new packets.

The OpenFlow's flow abstraction was designed to allow for some freedom regarding packet handling but still allow for an efficient hardware implementation. This was an essential tradeoff because too much flexibility would make it harder to create an efficient implementation, while the lack of flexibility would make it unsuitable for real world usage. Another important aspect regarding OpenFlow is that it was designed to work using already existing hardware, making its adoption simpler [28].

2.3.3 Scalability

The ability to control the network as a whole and the concept of "logical centralization" often leads one to believe that SDN implies a centralized architecture. That is not true, and indeed several distributed controllers already exist.

Considering that networks can have hundreds or thousands of network elements and

that the network is an essential infrastructure, relying on a centralized architecture would indeed be impractical. First, it would be a single point of failure, one that could compromise the entire network's ability to function properly. Second, managing a whole network is a resource intensive task, as such a single node wouldn't be able to scale to large networks. Several works (e.g, [24]) have shown that it is possible to distribute the control plane of an SDN, and by doing so achieve the necessary fault tolerance and scalability requirements to handle large networks.

2.3.4 VM migration in SDN

The flexibility of control offered by an SDN has been leveraged to enable novel strategies for VM migration. For instance, the work by Huandong Wang et al. [30] addresses this problem of scheduling migrations allocating network resources for migration, in a scenario where multiple VMs need to be migrated at the same time. Migrating multiple VMs at the same time sets different requirements on the network, when compared to a single VM per-step migration. For example, if two VMs migrate using the same path then the total migration time is increased as they compete for resources. Planning is thus necessary to avoid these negative interactions between different migration tasks. This work leverages parallel migration and multiple routing paths which help reduce total migration time. The authors adopt the network's perspective, aiming to reduce total migration time by maximizing the effective transmission rate in the network.

They formulate this as a mixed integer programming (MIP) problem, and propose an approximated algorithm (fully polynomial time approximation, FPTA) that aims to solve the problem in a more scalable way. By addressing this problem in an SDN environment the authors leverage the flexibility to install forwarding rules in order to provide multipath forwarding for migrations. They have shown that the one-by-one migration scheme was outperformed by all other solutions, with respect to total migration time. The approximated solution is considerably more scalable than the optimal, as the computation time is at most a polynomial function of the number of the migrations, taking significantly less time than solving the MIP problem.

The work by Xibo Yao et al. [31] also addresses multiple VM migration with the goal to reduce the total migration time and the downtime, by fully utilizing network bandwidth resources. In this case the algorithm maximizes the migration bandwidth by minimizing the correlations between different migration schemes of VMs. As a result the performance of this scheme improves over FPTA.

2.4 Network virtualization

Similarly to computational resources, network resources have also been target for virtualization. Well known examples include VPNs and VLANs. Unfortunately, these

traditional forms of network virtualization lack the flexibility and scalability to offer full virtualization, at cloud scales, of topology, addressing, and service models.

The lack of full network virtualization hinders several aspects of datacenter management. For example, without it the workloads that depend on a specific topology require network changes for their deployment. Other issues include the address space. If the substrate network is IPv4 the user is restricted to this particular addressing scheme [23].

Ideally, the network should work much like a virtualized computer works: any network topology and addressing scheme would be valid and attainable to be deployed on a single physical network. This section presents works that aim to achieve this goal.

2.4.1 Network Virtualization in Single-Provider Datacenters

Recently, VMware has presented its SDN-based network virtualization platform, NVP [23]. NVP's architecture is based around a network hypervisor that provides network virtualization abstractions to tenants. Namely: a control abstraction, meaning that the tenants are able to create network elements (called logical datapaths) and configure them as they would in a physical network; and a packet abstraction, which means that any packet sent should be given the same treatment (i.e the same switching, routing and filtering service) they would have if this virtual network was the tenant's own physical network. The platform implements the logical datapaths entirely on the software virtual switches present in every host, leveraging tunnels between every pair of host-hypervisors for connectivity and isolation. As a result, the underlying physical network only sees regular IP traffic and no special infrastructure nor special network feature is needed.

To implement multicast features NVP constructs a multicast overlay using additional nodes, called service nodes. Because NVP targets enterprise customers some tenants will want to connect their virtual networks with existing physical networks. For this purpose NVP uses special nodes called gateways.

Due to NVP's reliance on software switching and the fact that a centralized SDN controller cluster configures all the virtual software switches, some problems need to be addressed. Namely, ensuring that the software switches can provide sufficient throughput, and that the controller can scale. To improve forwarding performance several techniques were used. First, NVP explores traffic locality by installing flow rules in the kernel (for faster matching). Second, STT (Stateless Transport Tunneling) is used for encapsulation, since STT places a fake standard TCP header to make possible the use of NIC hardware offloading mechanisms, improving forwarding performance. As the computations performed by the controller are complex, there was the need to allow for incremental computation. To accomplish this goal, a domain specific language was created, called nlog. Finally, NVP addresses scalability and availability. Since the forwarding state computation is parallelizable NVP divides such computation into multiple tasks that execute independently in different servers. Availability is achieved by having hot standbys that

come into action as soon a failure is detected.

2.4.2 Multi-cloud network virtualization

Whereas NVP [23] focused on a single cloud with full control of the hardware, the authors of [6] extend the idea to span multiple clouds, both public and private, with differing levels of control regarding the physical infrastructure.

The ability to use multiple clouds provides several benefits such as reduced costs (by moving some workloads to cheaper clouds) or improving performance (by bringing services closer together). To fulfill its requirements the platform leverages the SDN paradigm, using Open vSwitch (OvS) [29] as a software switch for virtualized environments). Given the lack of flexibility in public clouds (i.e., no access to the network hypervisor) the authors opt for a container-based virtualization[1] solution running on top of the substrate VM. To achieve full network virtualization, the platform's hypervisor translates the virtual events to physical events (and vice-versa) and performs flow translation at the edge.

Another related approach is XenFlow [27], a network virtualization system that provides isolation between virtual networks, preventing one network from disrupting other network's performance and providing intra-network and inter-network QoS provisioning using a resource controller. XenFlow's architecture consists of three main elements: the virtual routers, the XenFlow Server, and a packet forwarding module. It runs on commodity hardware using Xen and a packet forwarding module compliant with OpenFlow [28] (the packet forwarding can be handled by Open vSwitch [29]).

Virtual routers run the XenFlow client module. Its function is to monitor the routing table and ARP table for updates, and to collect this information. Then, the XenFlow server module gathers all information from all the xenflow clients, creating the Routing Information Base to be sent to the server for every virtual router. To ensure isolation between networks XenFlow uses the VLAN tag. By associating virtual routers with queues in Open vSwitch, XenFlow achieves resource isolation (memory isolation is accomplished using standard Xen primitives). The difference between standard OpenFlow queue control and XenFlow is that XenFlow redistributes the idle capacity to maximize link utilization. Quality of service is also achieved by mapping flows to queues.

VirtualWires [33] presents an abstraction that facilitates multiple-cloud network virtualization. The authors proprose a connect/disconnect primitive that allows the user to connect vNICs with a point-to-point tunnel. It's implementation guarantees that when two vNIC are connected, they will stay connected even if one machine is migrated away (even to another network, across clouds). The links between two vNIC are provided by *connectors* layer-2-in-layer-3 network tunnels. To allow connectors to work across clouds the authors propose another component, called the *extender*, that maintains a permanent VPN connection with all other clouds. These are invisible to the VMs and are only used if

packets need to transverse cloud boundaries. The management of the vNIC's connections is implemented at the hypervisor level and tied with the live migration mechanisms, so that the connection between two vNICs always remains connected. VirtualWires was implemented in Xen and leveraged Xen-Blanket [34] to deploy VirtualWires across multiple clouds.

2.5 Network migration

The work discussed in section 2.2 argued that migration of VMs grants benefits, lowering costs and improving efficiency. This section focus on a related, but more complicated problem: the migration of an entire network.

2.5.1 Scheduling for VM migration

VROOM [32] was probably the first solution to migrate networks. Specifically, VROOM is a network management primitive that allows virtual routers to freely move between their physical counterparts. It can help in tasks like planned maintenance and service deployment, and solves new problems such as reducing energy consumption. VROOM detaches the control plane of routers from the physical equipment, allowing both to be independent.

The solution requires the ability to migrate links and having the capability to migrate the control plane. To address link migration, advances in transport-layer technology were leveraged, allowing VROOM to change the physical links between physical routers just by signaling the transport network. To address the router control plane virtualization VROOM partitions the resources of a physical router into several virtual routers that run independently, each with its own control plane (i.e., configurations, routing protocol instances). To enable router migration, VROOM's virtual routers can dynamically change their binding with the router's physical interfaces.

The migration process consists of the following steps: First, a tunnel is setup between the two routers, so that the new location can start receiving routing messages (before link migration is completed). It's important to note that the data plane can keep forwarding packets during control plane migration, as they are separated. After the control plane is migrated the data plane is cloned. This process is done by the control plane, that recreates the data plane state using a unified interface that hides the heterogeneity between different data planes. After the control plane is migrated the links are migrated asynchronously, until all links are migrated. When that occurs the tunnel is torn down and the migration is complete.

The authors of [26] address a different problem, the scheduling a virtual network migration. This problem is similar to VM placement as different migration plans can affect the traffic, incurring in more/less overhead and increasing/lowering migration time. The proposal leverages on previous work on live migration of a single router without

disruption to present algorithms to find optimal single-node at a time sequence of moves to minimize network overhead.

In addition they proposed algorithms that migrate multiple nodes simultaneously. To address the scheduling problem the authors designed 3 algorithms. The Local Minimum Cost First (LMCF) algorithm attempts to minimize the migration cost and moves only one node at a time. The Maximal Independent Set-Size Sequence algorithms tries to minimize the migration time by migrating multiple nodes simultaneously. Finally, Maximal Independent Set-Local Minimum Cost First (MIS-LMCF) tries to minimize both migration time and cost. Their experimentation show that no algorithm consistently provides the lowest cost but they all generate migrations with reasonable costs and lengths.

In VROOM the temporary cloning of routers creates network correctness problems, an issue addressed by LIME [17]. In this paper, the authors first define transparent migration. Under this definition, a migration is transparent if the events that happen during a migration can also be observed when there is no migration. For example, networks can lose packets, therefore packet loss during migration does not break transparency. The system, proposed in the context of an SDN, is capable of efficiently migrating an ensemble (a collection of virtual machines and virtual switches), transparently for any given SDN controller and end-host applications.

The authors start from a *move* primitive that works in the following way. When it is necessary to move a switch, the original rules are copied to the new switch. When the new switch has a complete copy of the rule set, the old switch is ”frozen” (a high priority drop rule is installed in the switch) and tunnels are established between the connected hosts, from the old switch to the new switch. As a result, hosts connected to the old switch have their data sent to the new switch through a tunnel, while hosts connected to the new switch use it directly. This primitive ensures that transparency is preserved, but it is costly. For instance, if two local hosts want to communicate after the tunnel is setup, they have to use the tunnels twice, incurring huge latency (Figure 2.9). Also while the switch is ”frozen” it is unresponsive leading to high rates of packet loss. To avoid this problem, a different solution is proposed. Instead of having only one working switch at a time, there are multiple clones of a switch executing on multiple physical switches (similar to VROOM).

This avoids packet loss, and allows two hosts to communicate locally (Figure 2.10). This cloning solution has, however, a problem: it might cause incorrect SDN application behavior. To prevent this problem LIME merges events and statistics from all the running instances of a switch, in order to preserve the single switch view a controller application expects. This means merging packet-in events, traffic statistics, links failures and rule timeouts. It is important to note that this merging process cannot preserve the order of events, due to the variable delays in receiving control messages. Fortunately, this does not hinder transparency, because even a single switch does not preserve the order of

events (packets arriving on different linecards could trigger events in any order) and the OpenFlow specifications does not require it so no special care was needed.

Finally, LIME addressed the problem of updates during a migration. Because switches can't be updated at the same time (i.e. atomically), some application's inter-packet dependencies may be broken, leading to incorrect application behaviour [16]. To address this issue LIME proposes two solutions: The first solution works by installing a drop rule, and only when both switches notify that this rule has been installed does the controller install the updated rule. The end result is that packets are either dropped or processed by the new rule. The other solution relies on the fact that one switch can be updated atomically, while the other one simple detours the affected traffic to this first switch.

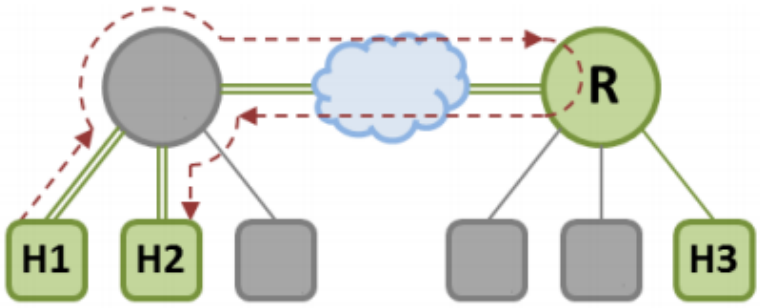


Figure 2.9: H1 and H2 could communicate locally, from [17]

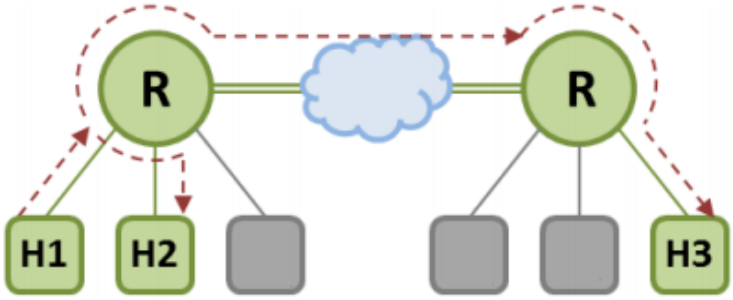


Figure 2.10: Local communication, from [17]

2.6 Final considerations

The study of the state of the art presented in this chapter was illuminating with respect to our problem. For instance, various techniques employed in the context of virtual machine migration can be leveraged and modified for the context of network migration. It was also clear that the affinity between certain system components (i.e. the amount of traffic exchange between entities) can be explored to build effective and efficient algo-

rithms.

This analysis survey also helped in understanding the challenges of network migration, particularly in the SDN-based multi-cloud network virtualization scenario we target. Specifically, the high latency and low throughput of cross cloud links, along with the fact that in an SDN the control plane is physically separated from the data plane, makes the problem more challenging.

Finally, to the best of our knowledge the problem of scheduling the migration of a software-defined network to fulfill the requirements of a multi-cloud scenario has not yet been addressed.

Chapter 3

Design and implementation

In the previous chapter we presented recent work that enables network migration for cloud environments. Unfortunately, none of these solutions addresses the challenges of a multi-cloud environment. In particular, no proposal to date has considered the problem of minimizing network disruption during migration across clouds.

For this reason, in this work we investigate and propose novel migration scheduling algorithms (also known as orchestration algorithms). The goal of these algorithms is to determine the order in which the set of nodes that compose a network should be migrated from one cloud to another. In particular, we address the problem of migrating a Software defined Network (SDN).

An SDN is different from a traditional network due to the fact that decisions of how to forward packets are not the responsibility of the devices that perform the forwarding (the switches). These decisions are made by a logically centralized element, the network controller.

In a reactive SDN (Figure 3.1), the most common type, the network elements have to communicate with the controller, to deliver network events for processing. The controller is then responsible for making the actual forwarding decisions, and then mandating the setup of flow state in network elements to implement these decisions. The communication between the network elements and the controller is typically done using the OpenFlow protocol [28], which runs on top of TCP or TLS. While this separation of control and data planes brings flexibility in networking, it poses some challenges.

The problem lies with the fact that the network switches are not independent to take decisions, relying on the controller for that purpose. Tasks like updating the network policy or reacting to new unspecified network events require communication with the controller. For this reason, it is of utter importance to maintain an as-stable-as-possible connection between switches and controller(s), avoiding disruption. Our multi-cloud environment is challenging in that respect, as it involves high-latency links. For this reason, a good migration plan is crucial, as the connection between the switches and the controller should be the least affected with the migration. In particular, the latencies should be kept

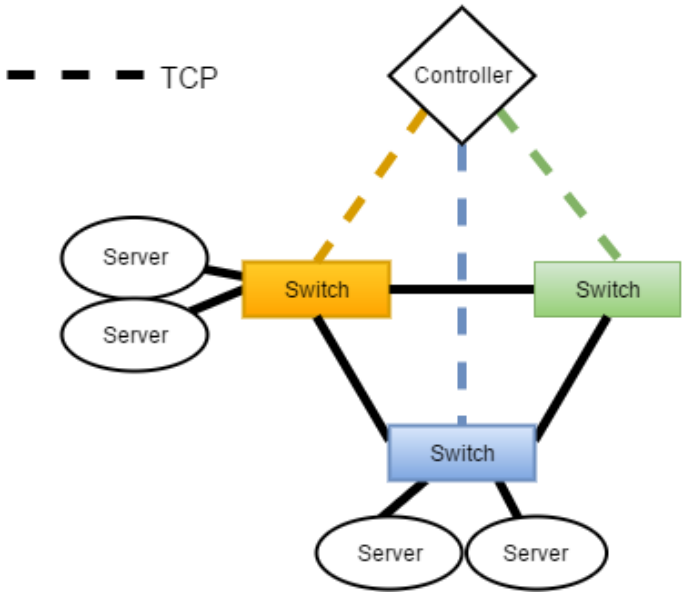


Figure 3.1: In an SDN all switches need stable connectivity with the network controller.

to a minimum, as otherwise there is the risk of breaking the TCP connections between controller and switches. If the ability to communicate with the controller is affected, so is network operation. In the setting of a single datacenter this problem is less urgent, due to the high-speed, low latency nature of the environment. By contrast, in a multiple cloud scenario, where the latency between datacenters is high, the problem gains a new dimension. A poorly assigned migration order can result in part of the network elements having to wait long periods to both receive messages from the network controller and taking longer to deliver network events to the controller, or even breaking the TCP/TLS sessions. Figure 3.2 shows an example of how a poor migration plan can affect the experienced latency significantly. The reader should note that in this figure, and in the rest of the chapter, when we represent the controller we are effectively representing the switch to which the controller is connected.

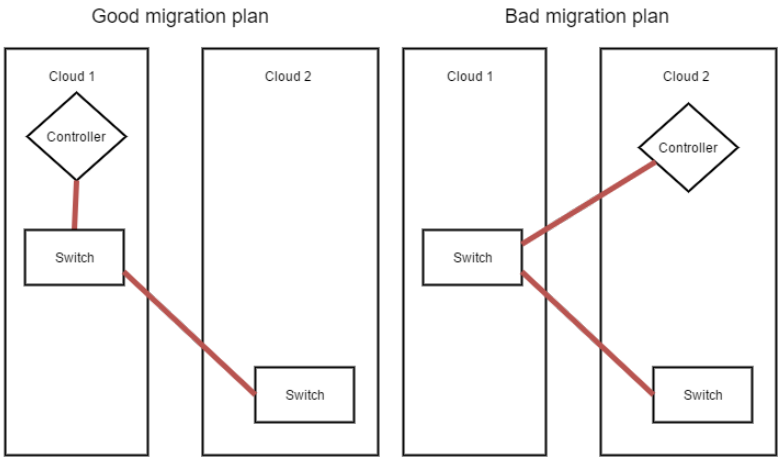


Figure 3.2: Avoiding bad migration decisions can result in less impact of control latencies

This figure assumes, for simplicity, a linear topology with three network elements: two switches and one controller. In this example, a bad migration plan (on the right) would consist in moving the middle switch first, resulting in the highest possible communication latency for the other switch (during the migration process): the edge switch would need two trips across a high latency link to reach the controller.

A different migration order, such as migrating the edge first (left figure) would result in an overall smaller impact on control communications (as it involves using the expensive inter-cloud links less often). This example clearly shows that the migration plan can mitigate disruptions during the migration process. The algorithms we propose next aim to find a migration order which minimizes control plane latency.

3.1 Context and objectives

After presenting the problem, we can now contextualize it. We start by describing the multi-cloud virtualization platform that will incorporate our solution. Figure 3.3 shows the architecture of this platform.

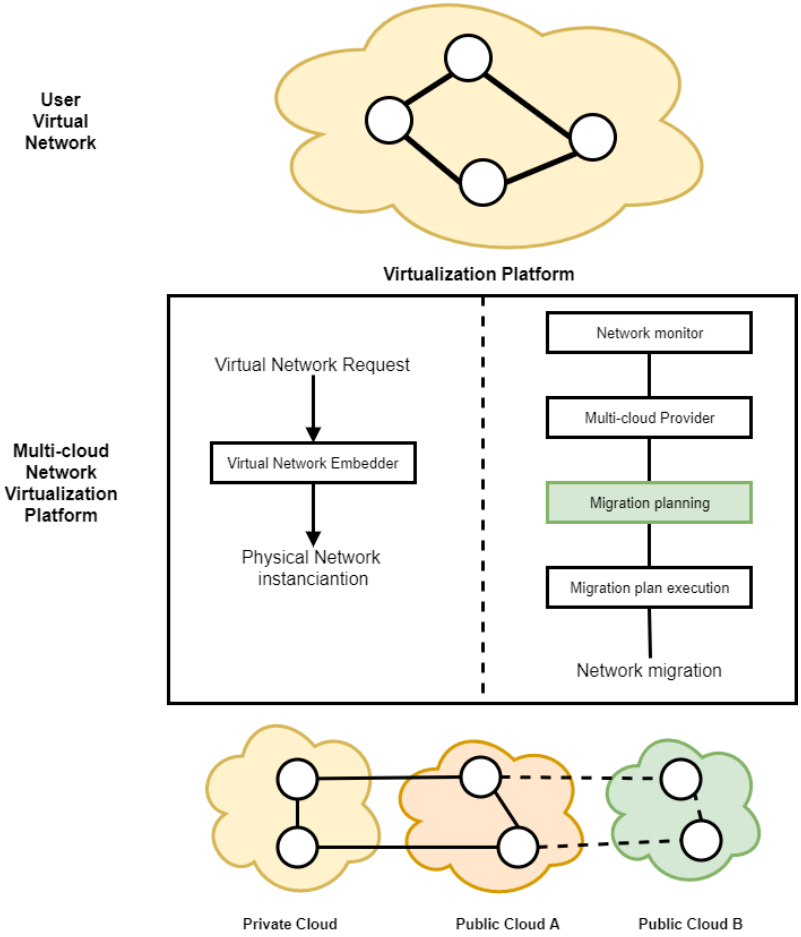


Figure 3.3: Multi-cloud virtualization platform and the network migration module

This platform (described in [5]) offers network virtualization to tenants across multiple public and private clouds. On the top we have the virtual network as seen by the user. This virtual network is then mapped by the virtualization platform into the substrate physical network. This is handled by the virtual network embedding module. To achieve several of its goals the platform requires migration capabilities. Amongst these, we highlight reducing costs (by migrating the network to a cheaper cloud), increase dependability (by migrating to a more resilient cloud service), or improving performance (by bringing network elements closer to each other and/or closer to users).

The implementation of these features requires a fundamental primitive: virtual network migration between clouds. We seek to contribute with this thesis with a sub-module of the migration module: migration planning (colored box).

The function of the migration planning module is to define the order of migration of every network element. The follow-up problem (its execution) can be solved by means of techniques as those proposed in [17]. As explained, our goal is to minimize control latency. Unlike optimizing for other metrics, such as bandwidth, to reduce control latency we can not rely on traditional techniques such as compression, since the network is unlikely to be significantly affected by the small control messages that are exchanged between the controller and switches. In contrast, migrating virtual machines and all their associated state is a good candidate for using compression techniques, and in fact has been considered in the past [13].

For our problem, however, orchestrating a good migration plan is key. A poorly designed plan can affect the latency significantly and lead to disruption in control plane communications by, for example, migrating nodes in a way that forces multiple round trips between clouds for an event to reach the controller, as explained before.

Next, we describe the several algorithms we propose to address the goal of minimizing control latency during network migration.

3.2 Proposed orchestration algorithms

This section describes solutions for the problem of finding a good migration plan. We start by presenting an optimal solution based on a linear programming formulation (MILP). As solutions based on mathematical programming have scaling limitations we present several heuristics that aim to achieve results as close as possible to the optimal while maintaining low execution times in larger networks.

3.2.1 Network migration model

Our network migration model is the following. For a given network with n nodes we consider that the migration process will take n stages, with one network element migrating at every stage, sequentially, with no delay between stages, until all the network elements

have been migrated. We do not consider the problem of migrating the virtual machines themselves – we consider these to be migrated by existing techniques [12].

3.2.2 Optimal strategy using Linear programming

In this section we propose a MILP formulation as an optimal solution for the problem. The goal is to minimize the average control plane latency. As such, the cost function can be represented as the sum of the network latency between the switches and their controllers at each stage (as we assume the number of elements does not change).

The formulation takes three inputs. The first consists of a $n * n$ matrix (n being the number of nodes in the network), containing the costs of links between every node. The second is the latency cost incurred in using links between clouds. Finally a matrix defining the edges make up the path each node takes to reach its controller.

The main variable that guides the search is x_{ij}, which holds the value 1 if the node n has been migrated at stage j of the migration process. Then, several constrains are added to ensure the variables only hold valid values (such as: a node can only migrate once).

We define the following parameters:

S - Set of nodes to be migrated.

n - size of set S (equal to the number of stages)

x_{ij} - Binary variable which holds the value 1 if the node i was migrated at stage j.

$c(i, j)$ - The cost associated with switch i at stage j. The cost is the latency between a switch i and the controller in stage j.

p_i - The set of edges that form the path from i to its controller.

$path(i, j, k)$ - binary parameter that takes the value 1 if the link between i, j is part of the path that k follows to reach its controller, takes the value 0 otherwise.

$not\_same\_network(a, b, j)$ - binary variable which takes the value 1 if the nodes a and b are not in the same cloud at stage j; 0 otherwise.

$normal\_path\_latency(a, b)$ - The cost that the path between a and b incurs when a and b are in the same cloud.

$tunnel\_path\_latency(a, b)$ - The cost that the path between a and b incurs when a and b are in different clouds.

$has\_migrated\_at\_stage(c, j)$ - This variable holds the value 1 if network element c has been migrated at stage j.

The variable $has\_migrated\_at\_stage(c, j)$ is defined using x_{ij}:

$$\sum_{i=0}^{j} x_{ij} \tag{3.1}$$

In other words, a network element has migrated at stage j if it has migrated in any previous stage.

The objective function is the following:

$$min \quad \sum_{j}^{n} \sum_{i}^{S} c(i,j) \tag{3.2}$$

The objective function aims to minimize the sum of every stage's sum of cost (latency) between switches and the controller. In other words, the objective function is to minimize the aggregated cost of every node to its controller at every stage of the migration process. As we assume only one node (switch or controller) is migrated at every stage, the following restrictions apply. First, each node is only migrated once:

$$\sum_{j=0}^{n} x_{sj} = 1, \forall s \in S \tag{3.3}$$

Second, at every stage only 1 node migrates:

$$\sum_{s=1}^{S} x_{sj} = 1, \forall 1 \leq j \leq n \tag{3.4}$$

$c(i,j)$ consists of the sum of the latency of every link in the path from i to its controller at step j. This cost function has two main components: intra-cloud latency and inter-cloud latency (given by the input parameters). Whether two network elements are in the same cloud or not is given by $not\_same\_network(a,b,j)$. The input parameter $path(i,j,k)$ defines which links make up the path each network element has to follow to reach the controller.

The cost of a given network element i at stage j is calculated as follows. For every pair of network elements. If this pair forms a link that i uses to reach the controller and the two elements in this pair are in the same network (given by $1 - not\_same\_network$) we incur in the intra-cloud latency cost (given by $normal\_path\_latency$) if this pair forms a link used by the network element i to reach the controller and the two elements of this pair are in different clouds then we incur in the inter-cloud cost (given by $tunnel\_path\_lantency$). If the pair does not make up a link that is used by the network element i to reach the controller then it is not necessary to add any cost.

Using these parameters we can define the function c (the cost of a given node at a given stage). The cost of the node at a given stage is the sum of the latency incurred in that node's path to reach the controller. It's defined as follows:

$$\begin{aligned} c(i,j) = \sum_{a}^{S} \sum_{b}^{S} (&not\_same\_network(a,nb,j) * path(a,b,i) * tunnel\_path\_latency(a,b) \\ &+ \\ &(1 - not\_same\_network(a,b,j)) * path(a,b,i) * normal\_path\_latency(a,b)) \end{aligned} \tag{3.5}$$

In other words, the cost of a node at a given stage, is calculated by considering every pair of nodes, removing those which the current node does not use to reach the controller (this is given by multiplying with $path$, which holds the value 0 if the given pair is not used by the node to reach it's controller), and then incurring on the necessary latency cost depending on whether the link goes across cloud boundaries or not.

As explained above, $not\_same\_network(a, b, j)$ is a variable that has the value 1 if a and b are not in the same network at stage j, 0 in case they are. The definition of $not\_same\_network$ is the following:

$$not\_same\_network(a, b, j) = |has\_migrated\_at\_stage(a, j) - has\_migrated\_at\_stage(b, j)| \tag{3.6}$$

In other words, two network elements are not in the same network/cloud if their migration status is different. If both network elements have yet to be migrated then $has\_migrated\_at\_stage$ holds the value 0 for both network elements.

This definition of $not\_same\_network$ has a problem: the absolute value function is not linear, therefore it's necessary to work around it so that it can be used in a MILP formulation. Solving this issue is done at the expense of aditional constraints. Our solution added four additional constraints. The first two are:

$$not\_same\_network(a, b, j) >= \sum_{i=1}^{j}(x_{a,i}) - \sum_{i=1}^{j}(x_{b,i}) \tag{3.7}$$

$$not\_same\_network(a, b, j) >= \sum_{i=1}^{j}(x_{b,i}) - \sum_{i=1}^{j}(x_{a,i}) \tag{3.8}$$

These two constrains (3.7, 3.8) force the variable to hold the largest value of the two possible values of the differences between a and b. These constraints alone, however do not solve the problem. If different values are assigned to a and b the outcome is correct, as both pairs (1,0) and (0,1) result in 1. However, when a and b are equal the result is not guaranteed to be correct. For example, with both variables set to 0 the previous constraints only forces the value to be larger than 0. As such, the result could be set to the incorrect value of 1 if it helps lowering the cost.

To address these issues, the following constraints where put in place:

$$not\_same\_side(a, b, j) >= 2 - (x_{a,s} + x_{b,s}) \tag{3.9}$$

$$not\_same\_side(a, b, j) >= x_{a,s} + x_{b,s} \tag{3.10}$$

The last constraint is variable x, which is binary variable:

$$x_{a,b} \in \{0, 1\}, \forall a, b \in S \tag{3.11}$$

3.2.3 Heuristics

A linear programming solution as the one just proposed achieves optimal results, but has have an important problem: it does not scale. We indeed show this in the evaluation of our proposals in Chapter 4. As the size of the problem increases the time to find its solution increases exponentially. Obtaining a solution to a problem in practice has a time limit, after which the solution is of no use. As it is important that our solution is viable in larger networks, it is necessary to develop heuristics: solutions that are not necessarily optimal, but that have acceptable execution time far lower than the optimal solution. Of course, a good heuristic should reach solutions close to the optimal. By analyzing the results obtained by our linear programming solution, we have attempted to deduce some intuition about the optimal migration orders orchestrated, with the aim to create heuristics that leverage this knowledge to reach a good solution, within a reasonable time frame.

3.2.4 Baseline algorithms

We first considered four algorithms that can be considered as baselines.

Random - A random migration order.

Controller first - Consists of sending the controller first, then randomly sending the switches.

Controller last - Consists of randomly sending the switches, and then the controller.

Controller in the middle - Consists of randomly sending half of the switches, then the controller, followed by the other half.

The purpose of testing baseline solutions is to provide an upper bound for the others, while the linear programming solution provides the lower bound (optimal result).

3.2.5 Lowest degree first

One important factor that is responsible for the increase in cost of a given migration plan is when a node (or group of nodes) is forced to follow a path to the controller that bounces between clouds several times (as seen in Figure 3.2). Nodes that are more likely to cause this problem are those that are connected to many other nodes (i.e, the high degree nodes). An example of such node would be a firewall, as it is a mandatory passage point in most networks. Migrating these nodes first is likely to cause a large disruption in the network, as several connections will have to cross between clouds to reach these nodes, and bounce back to reach its destination, greatly increasing the cost of the migration. As such, the first heuristic we propose works by migrating the nodes with lowest degree first, therefore trying to minimize the problem described above. Note that in our heuristic when a node is migrated it ceases to be considered as a neighbor for the degree calculations (lowering the effective degree for the remaining nodes). Our expectation is that nodes with high degree initially will see its degree decrease to the average as their neighbors are

migrated, thus reducing the impact of their future migration.

In case of a tie nodes that have a connection to a previously migrated node are prioritized. In other words, if two nodes have the same degree, but one of them has a connection to the previously migrated node, then the node with such connection is chosen first. The idea is to prefer migrating intermediate nodes that we are sure will entail a long communication path.

Definitions	
$nodes$	Set containing all network elements
$calculate\_degrees(topology)$	Returns a list of nodes sorted by degree
$topology$	Structure containing the whole topology
$degree(x)$	Returns the degree of a node x
$exists(t, set)$	True if t exist in set
$connected(a)$	Returns a list of a's neighbors

Table 3.1: Functions and variables used in the smallest degree first heuristic.

Algorithm 1 Lowest degree first

1: $order =< emptylist >$
2: $dangling\_nodes =< emptylist >$
3: **while** $\#nodes > 0$ **do**
Find the network element with the lowest degree
4: $targets = calculate\_degrees(topology)$
5: $target = targets[0]$
Within the network elements with the same degree find one who has a connection with the previously migrated element.
6: **for** $node\ in\ targets$ **do**
7: **if** $degree(node) == degree(target)$ and $exists(target, dangling\_nodes)$ **then**
8: $target = node$
9: $break$
10: **end if**
11: **end for**
12: $dangling\_nodes =< emptylist >$
13: $order.append(target)$
14: $nodes.remove(target)$
Keep track of the network elements that $target$ node is connected to so that we can in the next iteration verify if some low degree node has connections to the previously migrated node.
15: **for** each node n in connected(target) **do**
16: $dangling.append(n)$
17: **end for**
18: **end while**

Algorithm 1 works as follows (Table 3.2 shows how the degree changes every stage as nodes are migrated for the example topology). First, find the node with the lowest degree. Then, search for all the nodes with the same degree. If we find a node with the same degree and that also had its neighbor migrated in the previous step, then that node is the one to be migrated. The process repeats itself until there are no nodes left to be migrated.

Stage	A	B	C	D	E	F	G	Migrated
1	1	4	2	4	3	2	2	A
2	X	3	2	4	3	2	2	G
3	X	3	2	3	3	1	X	F
4	X	3	2	3	2	X	X	E
5	X	2	2	2	X	X	X	B
6	X	X	2	2	X	X	X	D
7	X	X	2	X	X	X	X	C

Table 3.2: This table shows how the calculated degree for each node evolves as the migration process progresses. For the example topology, entries marked with a cross mean that that node is no longer a candidate for migration

Figure 3.4 shows an example of this heuristic. First, node A is migrated, due to its low degree of 1. Then, node G is migrated, due to its degree of 2 (could also be F). Afterwards node F is migrated. Node E is then migrated due to its degree and because it's a neighbour of F. And so on.

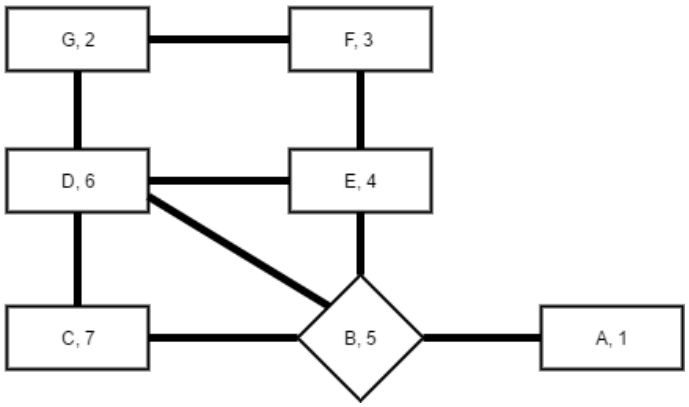


Figure 3.4: The migration order chosen by the lowest degree first heuristic. The numbers represent the migration order.

3.2.6 Bring me closer

This greedy heuristic seeks to minimize quickly the expensive cross-cloud links. For this purpose, when deciding to migrate a node it will favor nodes that, when migrated, will create the highest number of local (intra-cloud) links. Figure 3.5 shows an example of how a node is chosen for migration. Depicted in orange are the network elements that have already been migrated, in green the next element to be migrated according to this heuristic. For example in the top right figure the green switch is connected to two elements already migrated (controller and top-left switch) while the switches at the bottom are connected to only one or none.

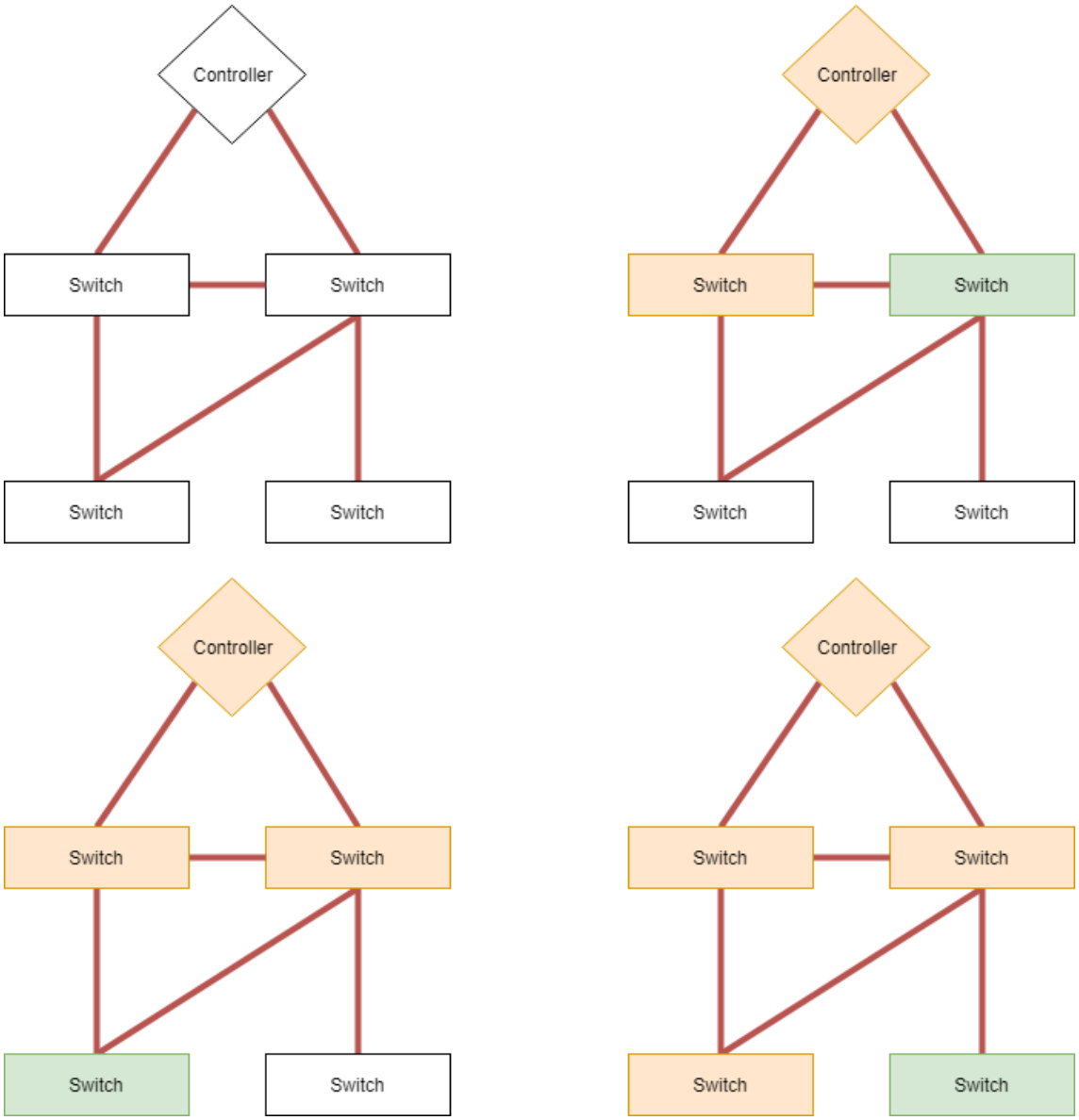


Figure 3.5: Bring me closer example. The green elements correspond to the next element being migrated

The heuristic is detailed in Algorithm 2. It works as follows: To determine which node to migrate we determine the node with the highest number of inter-cloud connections. To do so we iterate over every node and add one for every inter-cloud connection the node has. Then, we choose the node with the highest number of inter-cloud links to migrate. The process repeats itself until there are no nodes left to migrate.

Definitions	
$nodes$	Set containing all network elements
$dict()$	Key-value collection
$link(a, b)$	True if a has a link to b
$sort(x)$	Sorts a collection

Table 3.3: Functions variables used in the connection heuristic.

Algorithm 2 Bring me closer

```
migration_order =< emptylist >
nodes_to_migrate = nodes
while #nodes_to_migrate > 0 do
    inter_cloud_links = dict()
    for node t in nodes_to_migrate do
        for node d in migration_order do
            if link(t, d) and d exists in order then
                inter_cloud_links[t] += 1
            end if
        end for
    end for
    targets = sort(inter_cloud_links)
    order.append(targets[0])
end while
```

3.2.7 Scoring system

The last two heuristics we develop both rely on the same scoring system, so this section provides insight into how are the scores attributed. The idea is that some nodes are more important than others with respect to the network control plane. For instance, the controller is the most important element, as all control plane packets are sent to the controller. In a similar way, some switches are more important than others. A core switch, one that is topologically located in the center of the network and makes it possible for other switches to reach the controller, is more important than a leaf switch that no control packet traverses (except the control packets it generates). The idea behind this scoring system is the notion that the element importance (relative to others) might help the migration scheduling.

The scoring system works as follows. Every element in the network is given a score (including the controller). The score of an element (switch or controller) is given by the number of connections to the controller that stand on (transverse) this element. For example, the controller is always given a score equal to the number of nodes, since the controller handles all the switches. As should be clear, this score is not equivalent to the degree, as shown in Figure 3.6.

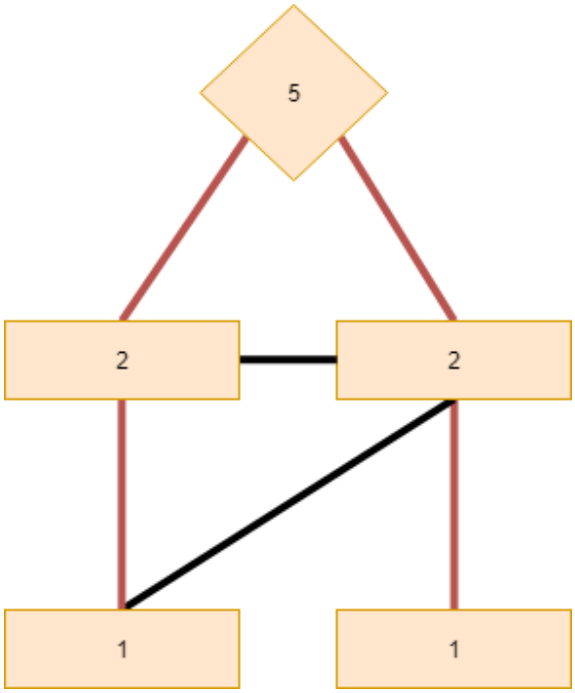


Figure 3.6: The red colored lines indicate the paths used by the switches to reach the controller. These are the ones taken into consideration when assigning the score to each network element. The number of each element represents its score. The score consists of the number of connections to the controller a given element supports (including itself)

Consider Figure 3.6, that shows an example of how to compute the score. The algorithm assumes shortest path routing to be used in the network, and the routing information to be available to the algorithm. As can be seen the controller receives control packets from all switches, and as a consequence is the element with the highest score: 5. The leaf switches only support themselves as such their score is the lowest possible. The core switches support both themselves and the leaf switches, hence have a score of two.

3.2.7.1 Boss in the middle

Testing all the candidate solutions isn't feasible considering real world time constraints. As such, we have considered scenarios such as migrating the controller as the first/last element, but quickly realized that such strategies produce poor results, as they would maximize the number of inter-cloud links right from the start of the process. Analysis of

the migration orders generated by the MILP solution showed that sending the controller somewhere around the middle of the process to be a good strategy. This offered some insight into how to reduce the search space.

The heuristic builds upon the score system explained above. The main idea is that the higher a node's score is, the closer to the middle of the migration process it should be. The controller (who has the highest score) should be migrated close to the the middle of the process (matching the intuition) and the leaf nodes who have the lowest score, should be amongst the first and/or last nodes to be migrated. In addition to this, we added some randomness to the process, as will be explained shortly. Figure 3.7 shows an example topology with the respective scores for the nodes.

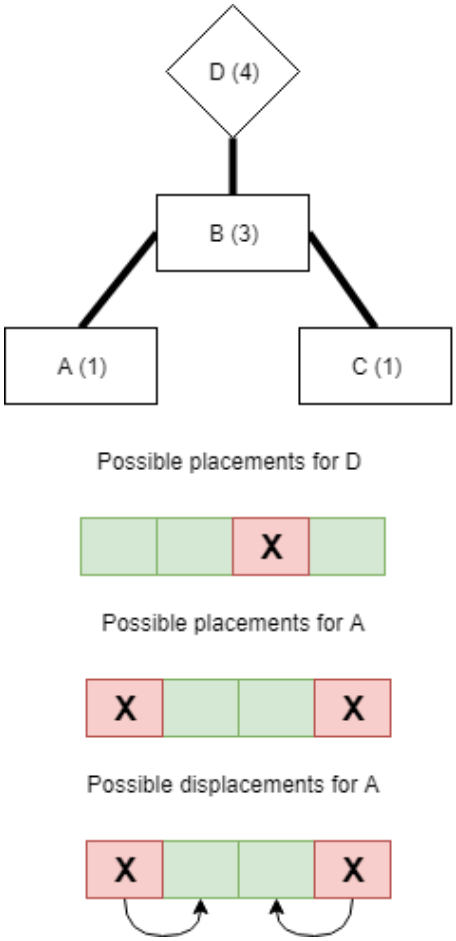


Figure 3.7: Core in the middle example. In red and crossed are the position pairs calculated for the labeled elements.

As explained above the intuition is that the controller should be migrated closer to the middle, hence it has a higher probability of being placed in the middle positions. By contrast, the controller has a lower probability of being placed at the start or at the end of the migration process. We thus start by placing nodes in their "right" migration

location, according to their score. However, we then include randomness to the process, by allowing nodes to be slightly displaced from their original positions. In figure 3.7, the initial position of D is in the center (given its high score) and of A in the edges (given its low score). But then these nodes can be slightly displaced, as is shown to A in the figure. This whole process is repeated multiple times and the solution with the best score is chosen.

The heuristic, in more detail, works as follows (required definitions in Table 3.4): First, we calculate the two possible positions for a given element (given by its score). To do so we determine the middle point of the migration order (line 2). Then, we calculate how many positions exist between the start and the middle point , d_left, and the end and the middle point, d_right (lines 4-5). Note that we assume integer division. To determine the first position, we convert the score of the node, which ranges from 1 to $\#nodes$ to the range of 0 to d_right. The first position is then given by $\#nodes - adjusted1$. For the second position we do the same for the range 0 to d_left.

After obtaining the positions for all elements in the network, according to this scheme, we pick one of the positions randomly, and then randomly skew it (up to a given maximum ratio, mr). That is, for every node we affect its score randomly up to a maximum number of places it can be moved (the "displacement"). The ratio is a number, between 0 and 1. The smaller the ratio the smaller the distance a node is allowed to be displaced to.

We perform this process a given k amount of times, and pick the best solution found. The metric we use to calculate the best solution is the $migration\_cost$, that is given by the sum of the control plane latencies from all switches to the controller, for all migration steps. Our assumption is that simply positioning the nodes based on their score may not be enough to achieve a good solution. As such, it is our expectation that by adding randomness and evaluating several candidate solutions the final result will improve.

Definitions	
$nodes$	Set containing all network elements
$sort(list, valueFunction)$	Sort $list$ with the values derived from valueFunction
$dict()$	A dictionary type (key-value table)
$scores$	A dictionary containing the key value pair of node-score.
$migration\_cost(order)$	Returns a numeric value of how good the given migration order is (lower is better)
$displace(number, ratio, n)$	returns $number$ randomly moved up to either side to a max of ratio*n
$choice(list)$	returns one randomly chosen element of the list

Table 3.4: Functions variables used in the algorithm.

Algorithm 3 Weighted sample

```
1: n = #nodes
2: middle = n/2 + 1
3: values = dict()
4: d_right = n - middle
5: d_left = middle
6: for node n in nodes do
7:     score = scores[n]
8:     maxNewRange1 = d_right
9:     adjusted1 = (score * maxNewRange1)/n
10:    maxNewRange2 = d_left
11:    adjusted2 = (score * maxNewRange2)/n
12:    pos1 = n − adjusted1
13:    pos2 = adjusted2
14:    values[n] = (pos1, pos2)
15: end for
16: LastOrder = []
17: LastScore =< verylargepositivevalue >
18: for int i in 1..k do
19:    order = sort(nodes, sortingfunc = displace(choice(values[n]), r, n)
20:    score = migration_cost(order)
21:    if score < LastScore then
22:        LastOrder = order
23:        LastScore = score
24:    end if
25: end for
```

3.2.7.2 Migration tree

The intuition behind the last heuristic we have developed is explained with the help of Figure 3.8. In this example, the migration plan that minimizes the latency is to migrate one of the switches first, then the controller, and finally the other switch.

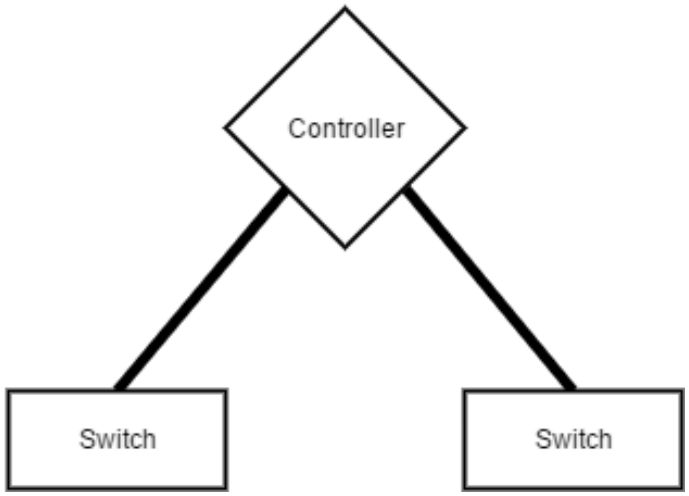


Figure 3.8: Migration tree simple example.

The migration tree heuristic works using the scoring system described in Section 3.2.7. The score system works on a virtual graph that includes only the edges that are used for control plane communication between the switches and the controller. Using this virtual graph and the principle explained above (Figure 3.8) this heuristic does the following. To migrate a certain node A we look at the neighbors (from the virtual topology) and divide them into two groups. The groups are defined in such a way that the sum of the scores of the node in each group should be as similar as possible. Then we migrate one of the groups, then node A, and finally the other group. By applying this idea recursively we reduce the problem of migrating a node with several neighbors down to the scenario shown in Figure 3.8, which consisted of only one controller and two neighbors.

Figure 3.9 shows how the heuristic works recursively on a larger topology. The heuristic begins at the controller, and divides its neighbors into two groups whose sum of scores are as similar as possible. The resulting two groups are the top-most element in the orange section, and the two top-most elements in the green section. Migrating the top-most element in the orange section requires the same process to be applied again. The process is applied recursively, ending when we reach a node that has no valid neighbors (i.e., no neighbors or all neighbors have already been migrated).

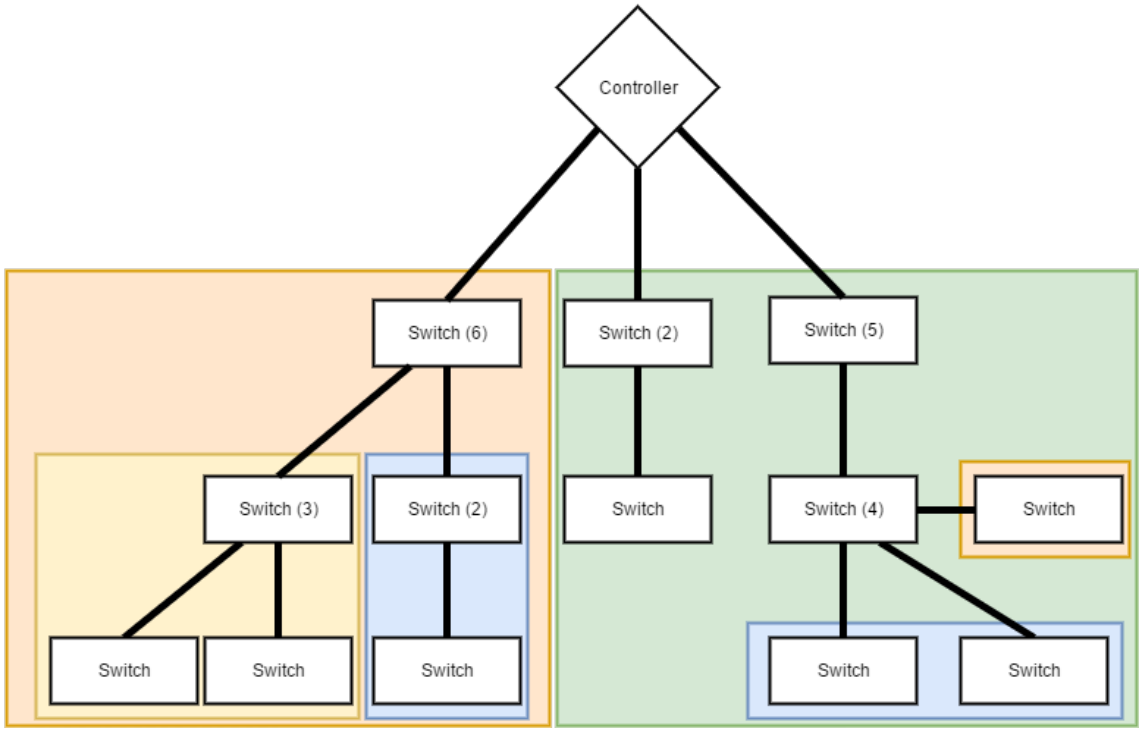


Figure 3.9: Migration tree. this figure shows how the heuristic recursively returns to the simple case shown in Figure 3.8. The color scheme represents how the recursion evolves (migrate one group, migrate node, migrate other group) until reaching the base.

The heuristic is presented as algorithm 5, and works as follows. In the first call of the recursion, $visited$ is an empty list, and lst contains only the controller. Firstly, all nodes are added to the $visited$ list, since they will be processed. Then, for every node in lst (the list of nodes we must process), we divide its neighbors into two groups, making sure the sum of scores of the nodes in these two groups is as similar as possible (algorithm 4). The migration order is then one of the groups, followed by the central element, then the other group of neighbors. The process repeats recursively.

Algorithm 4 is the function used to split the group of neighbors into two of similar weight. The chosen algorithm to perform this splitting is the following: The input is the set of nodes and their scores. If there are only 2 elements then splitting is trivial, the split returns 2 groups each with one node. When there are more than 2 elements we first sort the list of nodes by score. Then, we iterate the list, and when we have a sum greater or equal to half of the sum of scores of all the nodes in the original list, we have the first set. The second set being the remainder.

Definitions	
$nodes$	Set containing all network elements
$split(x)$	Splits x into two groups with similar score sum
$sortBy(l, k)$	Sorts collection l using k to determine values, descending order
$get\_neighbors(node)$	Returns the neighbors using the virtual graph

Table 3.5: Functions variables used in the algorithm.

Algorithm 4 Splitting function

```
Split(elements):
if length(elements) == 2 then
    return ( list(elements[0], list(elements[1] )          ▷ Trivial case
else
    currentSum = 0
    firstList = set()
    secondList = set()
    for node n in elements do
        currentSum = currentSum + n.value
        if currentSum < sum(elements) then
            firstList.add(n)
        end if
        if currentSum >= sum(elements) then
            secondList.add(n)
        end if
    end for
    return (firstList, secondList)
end if
```

Algorithm 5 migration tree

```
migrate(visited, lst, scores) :
if lst == [] then
    # The base case for the recursion
    return []
else
    # Mark all these nodes as visited
    for node n in lst do
        visited.add(n)
    end for
    migrate_order = []
    for node n in lst do
      # Obtain the neighbours for the given node
        neighbors = get_neighbors(node)
      # Split them into two groups whose sum of scores is as close as possible
        part1, part2 = split(neighbors)
        part1 = sortBy(part1, scores)
        part2 = sortBy(part2, scores)
      # Apply recursion
        order1 = migrate(visited, part1)
        order2 = migrate(visited, part2)
      # Define the migration order
        migrate_order+ = order1 + node + order2
    end for
    return migrate_order
end if
```

3.3 Summary

In this chapter we proposed several solutions to the problem of scheduling migrations in an SDN-based, multi-cloud network virtualization platform. We presented a linear programming approach which will find the optimal solution for the problem. We also presented several heuristics that aim to solve the same problem within reasonable time. For each of the heuristics we've described the intuition idea behind it, and its design and implementation.

Chapter 4

Evaluation

In this chapter we evaluate our solutions for planning the migration process in the context of multi-cloud network virtualization. We have compared the different algorithms using different topologies – linear, ring, random, and tree – to understand the trade-offs involved in each.

4.1 Environment Setup

The linear, ring and tree topologies were generated through python scripts using the default pseudo-random generator to assign latencies to the edges (numbers between 1 and 20, with connections that go across a cloud incurring an extra cost of 20) and to decide which element is the controller. The random topologies where generated with the gt-itm [2] tool. Each test run for a given topology with a given size was ran 200 times. The graphs presented in the next section show the average and the standard deviation of these 200 executions. The *boss in middle* heuristic is configured for 1000 rounds and a displacement of 0.1. We vary these parameters in section 4.6.

4.2 Linear topology

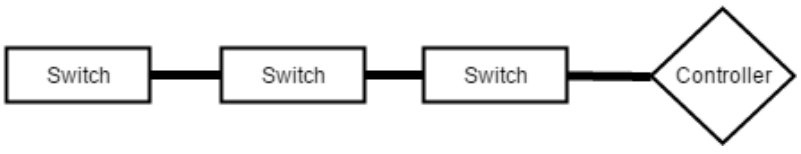


Figure 4.1: Example of a linear topology

A linear topology consist of n nodes arranged in a line. In our SDN setting one of the nodes is the controller (as shown in Figure 4.1). This node is randomly chosen.

Figure 4.2 shows the results for the various heuristics and the linear programming solution, when planing the migration of a network on a linear topology. Note that we only

show results for the optimal solution for networks up to 10 nodes, as the solution does not scale further.

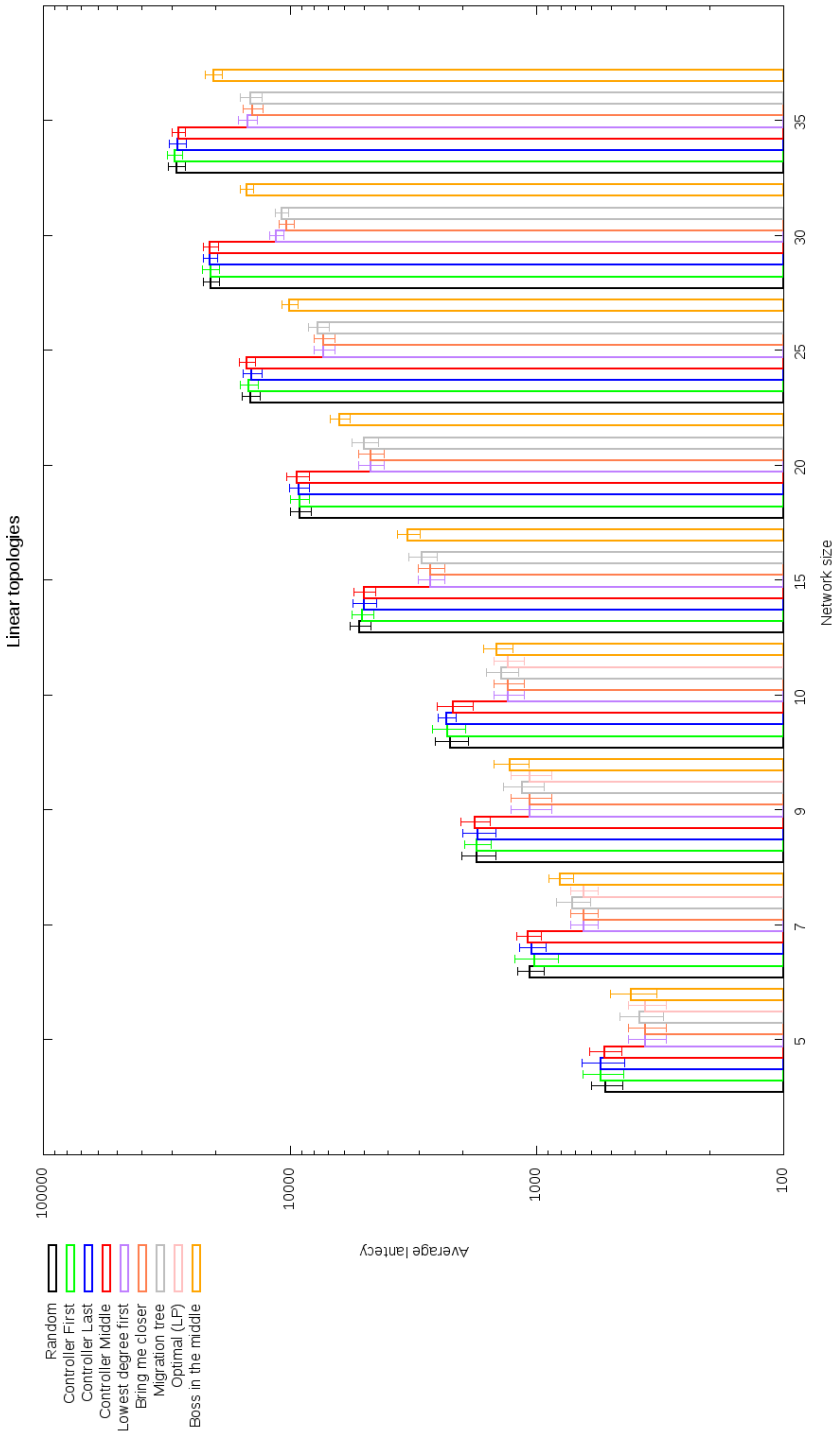


Figure 4.2: Switch-Controller latency: Linear topology

As expected, the random solution achieves very poor average latency to the controller

during the migration process. Sending the controller first or last is also a poor solution. These algorithms only work well enough when the controller is at one of the edges of the network. When the controller is not at the edge the number of cross-cloud connections increases significantly. In fact, by analyzing the migration sequences produced by the linear programming solution we found that only around 10% of the run the controller was migrated in either the first or the last step. If the optimal linear programming solution almost never adopts this strategy, then it is clearly a poor heuristic.

All other heuristics with the exception of the *boss in the middle* one, perform nearly as good as the linear programming solution and present an average switch-controller latency that is half that of the baselines. The *lowest degree* heuristic always chooses a good starting node: one of the edges, since they have the lowest degree. Also, the fact that this heuristic then prefers to migrate nodes that shared a connection to the previous node, make it perform well in a linear topology. The *bring me closer* heuristic ends up acting in a similar fashion, as it will try to migrate nodes that create the most local connections. Due to the fact that the topology is linear, attempting to create more local connections will migrate the nodes in a linear fashion. The *migration tree* heuristic performs along the same lines.

The *boss in the middle* solution performs worse in these topologies, due to the fact that this heuristic favors having the controller closer to the middle. In the cases when the controller is close to the edges, the heuristic will still try to have the controller around the middle of the migration order, which in these topologies is a poor fit.

In conclusion, the linear topologies are relatively simple to migrate therefore most heuristics we developed perform well.

4.3 Random topology

Random topologies (example in Figure 4.3) have no predefined shape. We generated them using the gt-itm [2] tool. These topologies should provide better insight to our solutions as many real world networks have this characteristic. Their random nature is challenging as the heuristics can't easily leverage the network structure to create a good migration order.

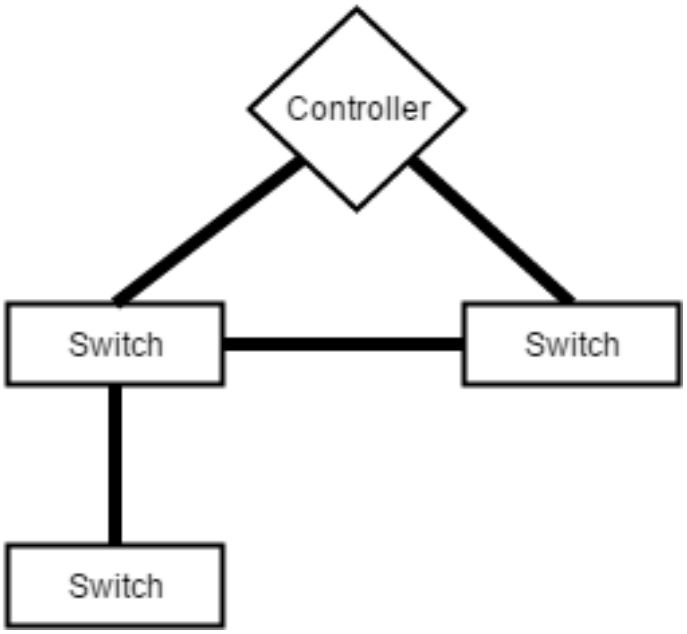


Figure 4.3: Example of a random topology

Figure 4.4 shows the results for all our solutions, considering random topologies. Similarly to the case with linear topologies, random heuristics perform poorly. As anticipated, sending the controller as first/last is not a good strategy. Interestingly, the strategy that places the controller at the middle of the migration is the best of the baseline solutions for random topologies.

The *lowest degree heuristic* performs better than the baseline solutions. Having a higher degree is shown to have some importance. This is common for various graph-based problems, as the degree if often a measure of the "importance" of a node. Again, our more sophisticated heuristics that take control plane latencies into account perform better than the others. In particular, the *migration tree* heuristic presents the best performance. The reason is mainly the fact that the *lowest degree* heuristic and the *bring me closer* heuristic are based on local actions and lack a "global" strategy to migrate the topology as a whole. The metrics these heuristics consider when calculating if a node should be migrated provide a good idea of the value of the given node , but the result of employing these decisions does not necessarily add up to a good overall strategy. On the other hand the migration tree heuristic works more "globally". The *boss in the middle* heuristic performs better than most heuristics (except for the *migration tree*). Both heuristics use the scoring system described above, demonstrating the value of this metric. The reason why *boss in the middle* is always consistently behind the *migration tree* is because it consistently attempts to place the controller in the middle, which is not always the best strategy.

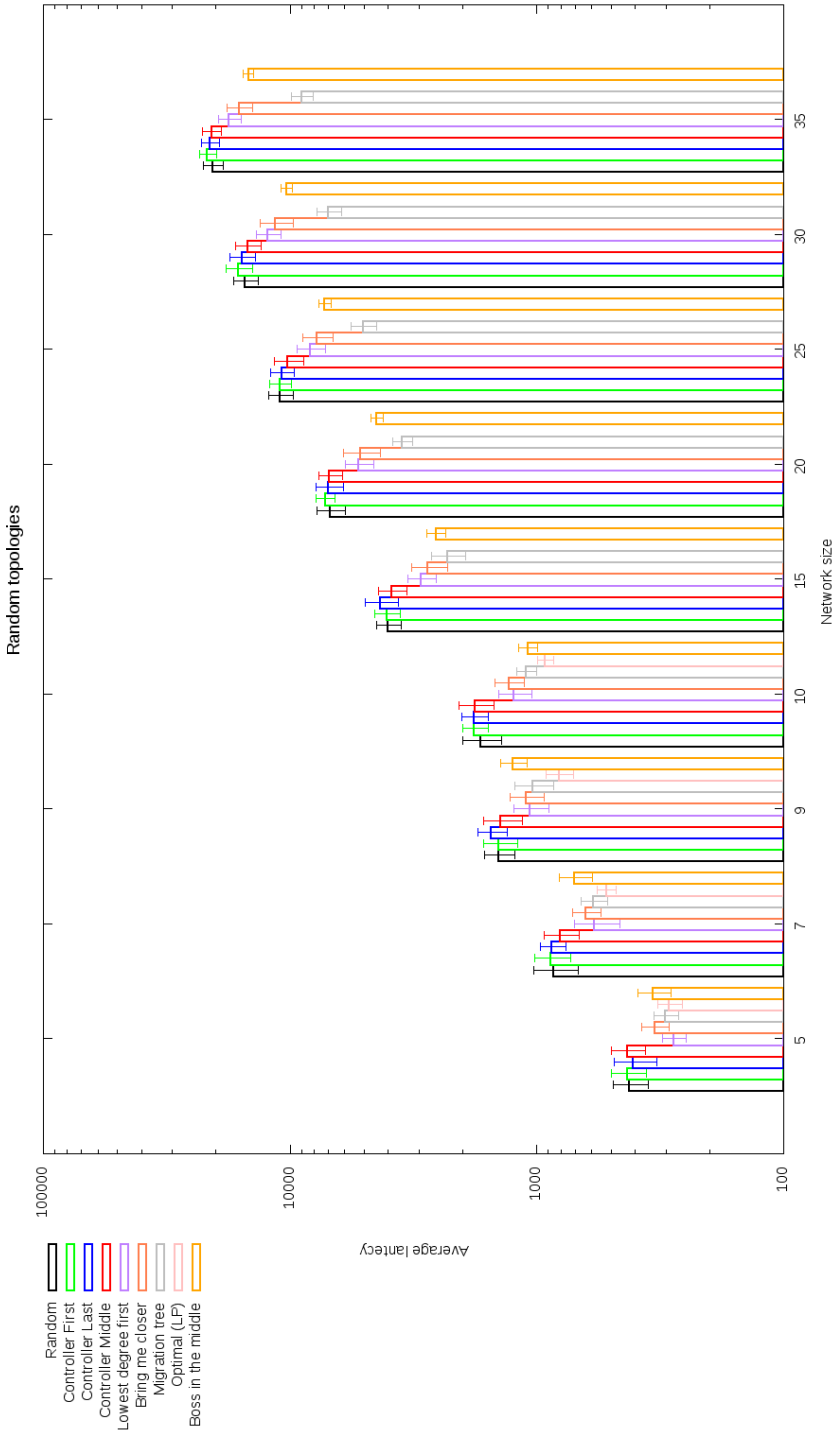


Figure 4.4: Switch-Controller latencies: Random topologies

4.4 Ring topology

Figure 4.5 shows an example of a ring topology. This topology is similar to the linear topology, with the difference that the first node is connected to the last. The results for this topology are presented in Figure 4.6. All heuristics perform similarly to the linear topologies. The baseline heuristics such as controller first/middle, perform again poorly, strengthening the conclusion that the position of the controller alone in the migration sequence isn't enough to reduce the overall experienced control latency.

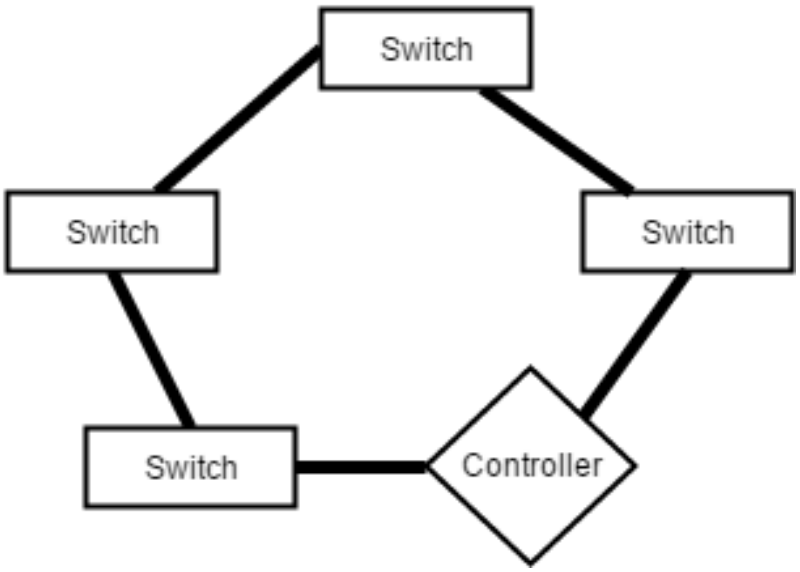


Figure 4.5: Example of a ring topology

The reason why the *boss in the middle* heuristic falls behind is due to this heuristic placing the controller in the middle. The intuition behind this heuristic that nodes with a low score should be placed close to the edges is mostly wrong in a ring topology (and in linear topologies as well).

The *lowest degree first*, *bring me closer* and *migration tree* heuristics all perform well and close to the optimal, for the same reasons as for the linear topologies.

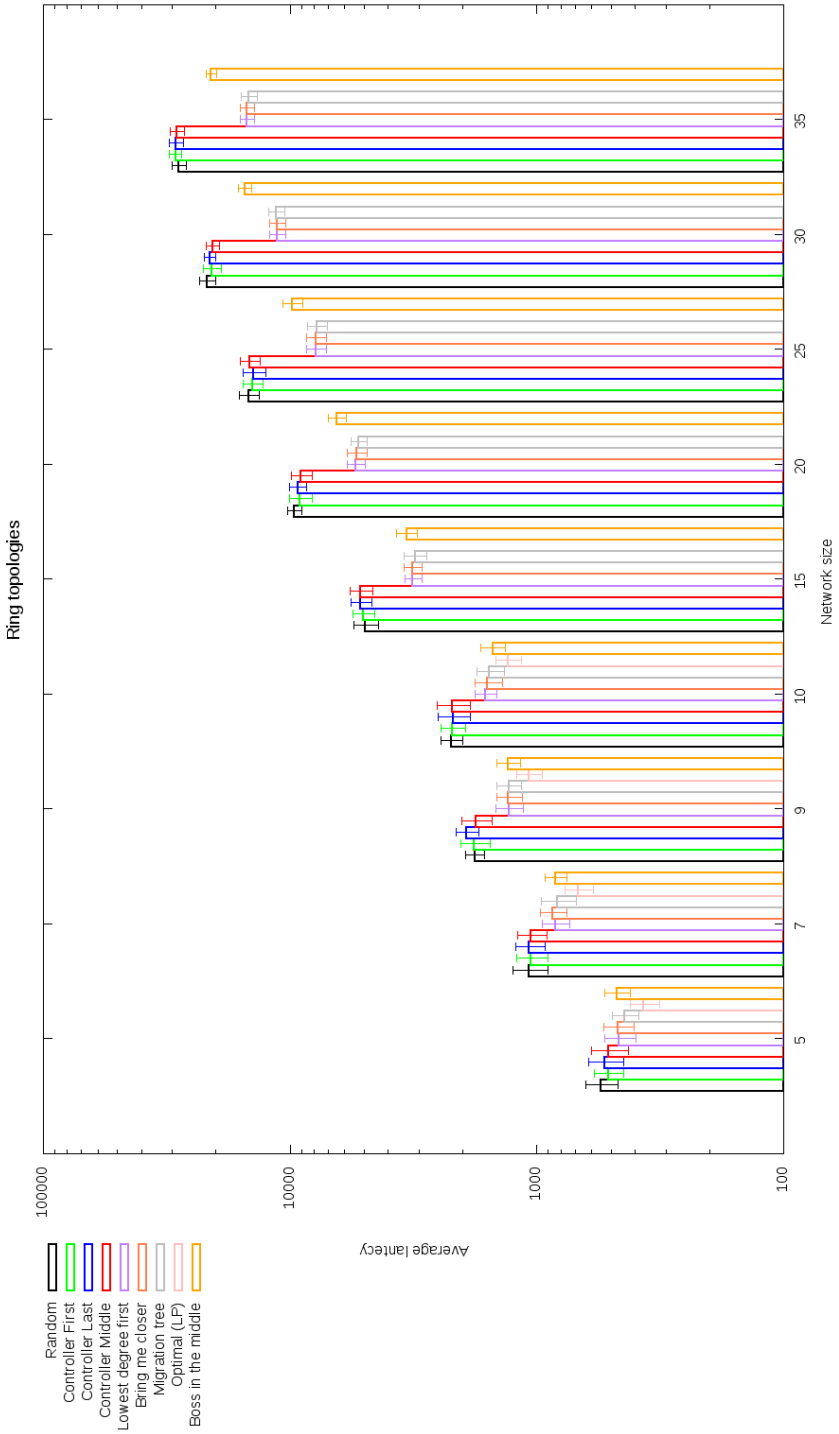


Figure 4.6: Switch-Controller latencies: Ring topologies

4.5 Tree topology

The last topology on which we evaluate our algorithms is the tree. Figure 4.7 shows an example of a tree topology. Note that the controller is chosen randomly it is not always the top-most element in the tree. The topologies generated consist of perfect binary trees (all interior nodes have two children and all leaves have the same depth or same level) of a given depth.

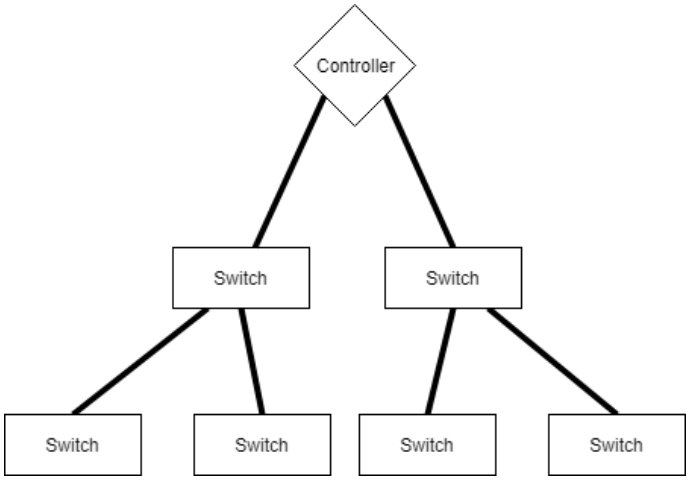


Figure 4.7: Example of a tree topology

Figure 4.8 shows the results for the tree topologies with n depth (in other words, perfect binary trees with n depth). Two algorithms stand out: the *lowest degree first* and the *migration tree* achieve results closer to the optimum. The former is good because in a tree it is a good strategy to start with the leaf nodes — which are the ones with lowest degree. The latter is naturally favoured by the tree structure.

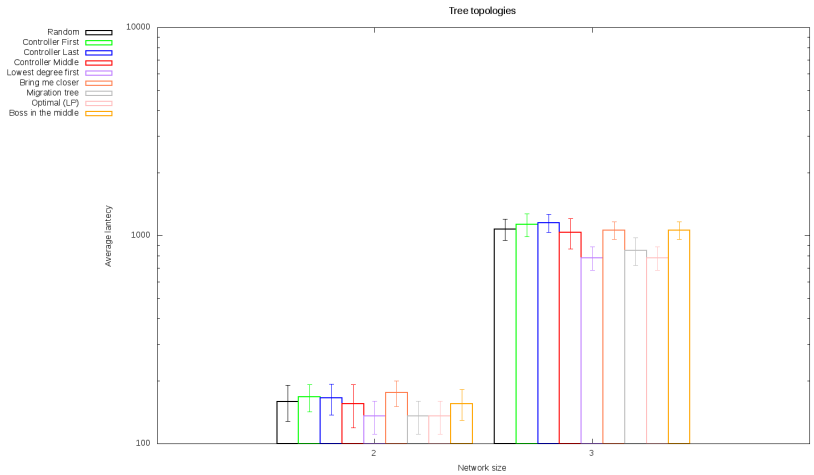


Figure 4.8: Switch-Controller latencies: tree topology, the x axis represent the depth of the perfect binary tree

4.6 Boss in the middle comparison

The *boss in the middle* heuristic has two input parameters: the move ratio and number of rounds. The move ratio is the maximum displacement. In other words, it is the maximum a node can have its position in the migration order moved. The number of rounds is the number of migration orders that are generated and evaluated before the best is chosen.

Figure 4.9 presents the results for random topologies using the *boss in the middle* heuristic with different input parameters. As expected, when using a lower number of rounds, the quality of the results decreases, since a smaller number of solutions are evaluated, the results are worse. Regarding the move ratio, running the *boss in the middle* heuristic with smaller move ratios provided better results, for the same number of rounds. This is an interesting result: The initial guess only improves with small displacements. Otherwise, an initial good choice is lost and the results become close to random.

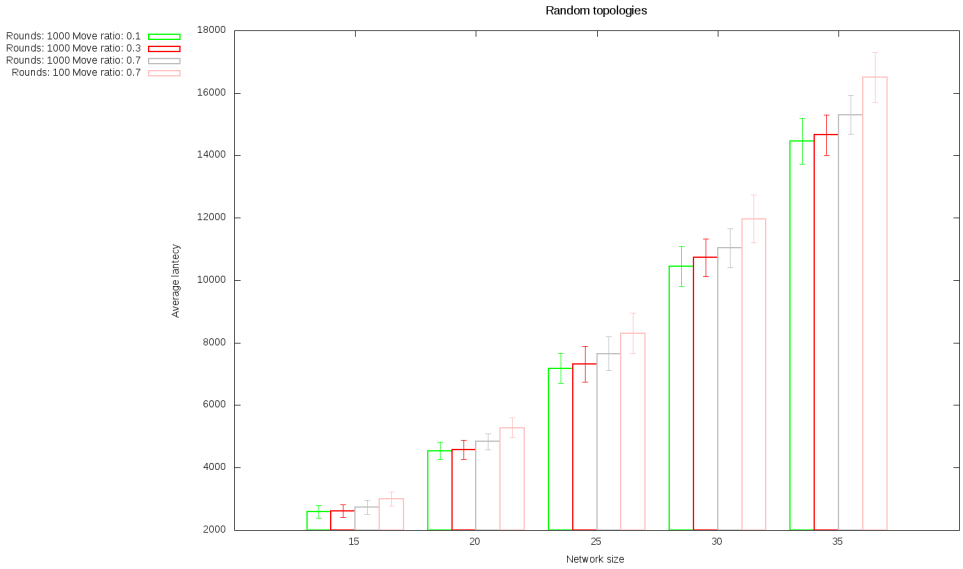


Figure 4.9: Results for several runs of the *boss in the middle* heuristic for random topologies, with varying inputs (move ratio, number of rounds)

4.7 Execution times

The execution time for the several algorithms are shown in Figure 4.10. As expected, the linear programming solution is orders of magnitude slower than the other heuristics. The execution time of the *boss in the middle* heuristic is also higher than the others because it needs to perform several rounds to chose the best result. The other algorithms are 10x to 100x slower than the baseline algorithms. Still, they are fast enough to remain competitive.

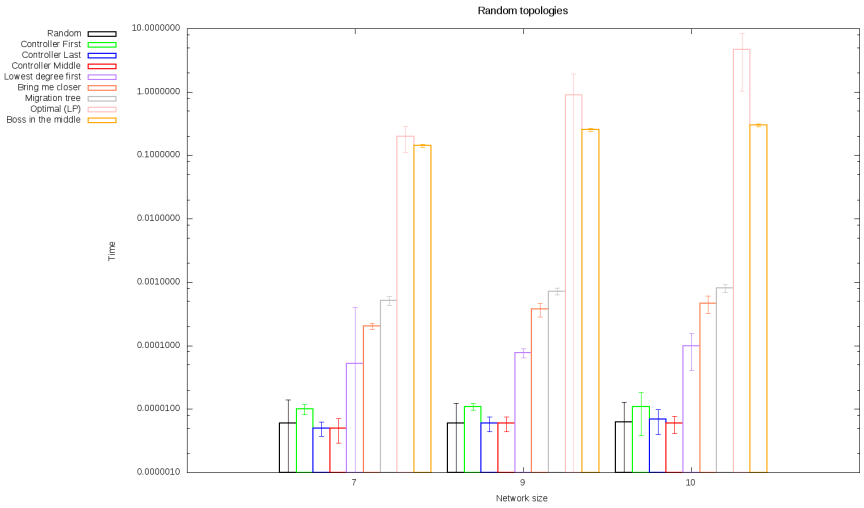


Figure 4.10: Execution time graph.The Y axis's scale is logarithmic.

4.8 Summary

In this chapter we evaluated the proposed solutions under several topologies – linear, ring, random, and tree. We analyzed the performance of these heuristics in the different topologies and how far the results were from the optimal solution. Overall, the baseline heuristics performed poorly. We also observed that some heuristics where favored by certain topologies, as seen by the results of the *lowest degree first* heuristic, which achieved good results for the linear topology but fell behind in the random topology. As main conclusion, the *migration tree* heuristic seems to be the best solution. Not only does it gives the best results for all topologies, achieving results several orders of magnitude better than the baselines and very close to the optimum, with very favorable execution times.

Chapter 5

Conclusion

Recent advances in networking, namely with the emergence of Software-Defined Networks, have led to the development of production-level network virtualization platforms. A central point of these platforms is to promote elasticity, flexibility, efficiency, and the ability to migrate workloads, including both the compute nodes and the networks that interconnect them. So far, existing SDN-based solutions have targeted a single provider, single cloud environment. As such, they are limited with respect to scalability and dependability, leading us to explore a solution that leverages multiple clouds for network virtualization.

To be able to perform network migration in a multi-cloud, SDN-based network setting, it is important to make sure that the connection to the controller is maintained stable during the migration. If the connectivity to the controller is affected, the network may take longer to respond to events, to implement network policies and, in the worst case, the connection between controller and switches can break. Therefore, it is crucial to orchestrate the way networks are migrated to ensure proper network operation. In this thesis we address this problem, proposing a linear programming formulation and several heuristics.

The goal of our algorithms was to minimize the control plane latencies, in order to avoid network disruptions. We have evaluated our solutions considering a range of typical network topologies. The main conclusion was that the solutions that considered the importance of a node, defined by the amount of control plane that traverses it, and that placed these nodes towards the middle of the migration, performed better.

There are several lines for future work. First, the algorithms we proposed could be implemented in a real multi-cloud network hypervisor (e.g., Sirius [6]). Second, the solutions could extend their goals from control plane latencies alone, to consider also the data plane traffic between compute instances to avoid bottlenecks.

Bibliography

[1] Linux containers: Why they're in your future and what has to happen first. http://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/openstack-at-cisco/linux-containers-white-paper-cisco-red-hat.pdf.

[2] Modeling topology of large internetworks. http://www.cc.gatech.edu/projects/gtitm/.

[3] Massive amazon cloud service outage disrupts sites, 2017 (accessed 20 April, 2017). https://www.usatoday.com/story/tech/news/2017/02/28/amazons-cloud-service-goes-down-sites-scramble/98530914/.

[4] Geetha Sowjanya Akula and Anupama Potluri. Heuristics for migration with consolidation of ensembles of virtual machines. In *Communication Systems and Networks (COMSNETS), 2014 Sixth International Conference on*. IEEE, 2014.

[5] M. Alaluna, F. M. V. Ramos, and N. Neves. (Literally) above the clouds: virtualizing the network over multiple clouds. *IEEE NetSoft*, December 2015.

[6] Max Alaluna, Eric Vial, Nuno Ferreira Neves, and Fernando Ramos. Secure and dependable multi-cloud network virtualization. In *1st International Workshop on Security and Dependability of Multi-Domain Infrastructures (XDOM0)*, April 2017.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5), October 2003.

[8] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.

[9] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[10] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, 2007.

[11] Martín Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*. PRESTO, 2010.

[12] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. NSDI, 2005.

[13] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. NSDI, 2008.

[14] J. Agogbua M. O'Dell J. McManus D. Awduche, J. Malcolm. Requirements for traffic engineering over mpls, 1999. RFC 2702.

[15] Soudeh Ghorbani and Matthew Caesar. Walk the line: consistent network updates with bandwidth guarantees. In *Proceedings of the first workshop on Hot topics in software defined networks*. HotSDN, 2012.

[16] Soudeh Ghorbani and Brighten Godfrey. Towards correct network virtualization. In *Proceedings of the third workshop on Hot topics in software defined networking*. HotSDN, 2014.

[17] Soudeh Ghorbani, Cole Schlesinger, Matthew Monaco, Eric Keller, Matthew Caesar, Jennifer Rexford, and David Walker. Transparent, live migration of a software-defined network. In *Proceedings of the ACM Symposium on Cloud Computing*. SoCC, 2014.

[18] Fang Hao, TV Lakshman, Sarit Mukherjee, and Haoyu Song. Enhancing dynamic cloud-based services using network virtualization. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, 2009.

[19] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven

wan. In *ACM SIGCOMM Computer Communication Review*, volume 43. ACM, 2013.

[20] Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. Supercloud: Opportunities and challenges. *ACM SIGOPS Operating Systems Review*, 49(1), 2015.

[21] Y. Rekhter K. Kompella. Virtual private lan service (vpls) using bgp for autodiscovery and signaling, 2007. RFC 4761.

[22] Eric Keller, Soudeh Ghorbani, Matt Caesar, and Jennifer Rexford. Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM WoXrkshop on Hot Topics in Networks*. HotNets, 2012.

[23] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, et al. Network virtualization in multi-tenant datacenters. NSDI, 2015.

[24] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. OSDI, 2010.

[25] Diego Kreutz, Fernando MV Ramos, P Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), 2015.

[26] Shihmin Lo, Moataz Ammar, and Ellen Zegura. Design and analysis of schedules for virtual network migration. In *IFIP Networking Conference, 2013*.

[27] Ferrazani Mattos, Diogo Menezez, and Otto Carlos Muniz Bandeira Duarte. Xenflow: Seamless migration primitive and quality of service for virtual networks. In *Global Communications Conference (GLOBECOM), 2014 IEEE*.

[28] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2), 2008.

[29] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15.

[30] H. Wang, Y. Li, Y. Zhang, and D. Jin. Virtual machine migration planning in software-defined networks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, April 2015.

[31] H. Wang, Y. Li, Y. Zhang, and D. Jin. Virtual machine migration planning in software-defined networks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, April 2015.

[32] Yi Wang, Eric Keller, Brian Biskeborn, Jacobus van der Merwe, and Jennifer Rexford. Virtual routers on the move: live router migration as a network-management primitive. *ACM SIGCOMM Computer Communication Review*, 38(4), 2008.

[33] Dan Williams, Hani Jamjoom, Zhefu Jiang, and Hakim Weatherspoon. Virtualwires for live migrating virtual networks across clouds. Technical report.

[34] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The xen-blanket: virtualize once, run everywhere. In *Proceedings of the 7th ACM european conference on Computer Systems*. EuroSys, 2012.

[35] T. Wood, K. K. Ramakrishnan, P. Shenoy, J. Van der Merwe, J. Hwang, G. Liu, and L. Chaufournier. Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines. *IEEE/ACM Transactions on Networking*, 23(5), Oct 2015.

[36] Kejiang Ye, Xiaohong Jiang, Ran Ma, and Fengxi Yan. Vc-migration: Live migration of virtual clusters in the cloud. In *Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on*. IEEE, 2012.